End-to-End Encrypted Group Messaging with Insider Security

by

Nik Unger

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Nicholas Hopper Professor University of Minnesota
Supervisor(s):	Ian Goldberg Professor Cheriton School of Computer Science
Internal Members:	David Jao Professor Cheriton School of Computer Science (Cross-Appointed)
	Sergey Gorbunov Assistant Professor Cheriton School of Computer Science
Internal-External Member:	Douglas Stebila Associate Professor Department of Combinatorics & Optimization

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

License

Statement of Contributions

The contents of Chapter 2 are partially adapted from a systematization of knowledge published in collaboration with Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith [UDB+15]. In particular, Sections 2.2, 2.3, 2.6, 2.7, and portions of Section 2.8 were jointly written with my coauthors based on group discussions. The contents of Sections 2.4 and 2.5 are partially adapted from a previous publication written in collaboration with Ian Goldberg [UG15], as well as my Master's thesis [Ung15]. I originally wrote the definitions related to deniability that appear in these sections based on a systematization of the existing literature.

The contents of Part II, including Chapters 5 and 6, were previously published in collaboration with Ian Goldberg [UG18]. I designed and developed the protocols and security proofs that appear in these chapters.

The remainder of the dissertation consists of newly written material. All of the work appearing herein was conducted under the supervision of Ian Goldberg.

The artworks appearing on pages xxiv, 10, 71, and 162 were commissioned for this dissertation and were illustrated according to my designs. The artworks were produced by Brandon Palas, Lukáš Vašut, Nicole Cardiff, and Tower Junkie Art, respectively.

Abstract

Our society has become heavily dependent on electronic communication, and preserving the integrity of this communication has never been more important. Cryptography is a tool that can help to protect the security and privacy of these communications. Secure messaging protocols like OTR and Signal typically employ end-to-end encryption technology to mitigate some of the most egregious adversarial attacks, such as mass surveillance. However, the secure messaging protocols deployed today suffer from two major omissions: they do not natively support group conversations with three or more participants, and they do not fully defend against participants that behave maliciously. Secure messaging tools typically implement group conversations by establishing pairwise instances of a two-party secure messaging protocol, which limits their scalability and makes them vulnerable to insider attacks by malicious members of the group. Insiders can often perform attacks such as rendering the group permanently unusable, causing the state of the group to diverge for the other participants, or covertly remaining in the group after appearing to leave. It is increasingly important to prevent these insider attacks as group conversations become larger, because there are more potentially malicious participants. This dissertation introduces several new protocols that can be used to build modern communication tools with strong security and privacy properties, including resistance to insider attacks.

Firstly, the dissertation addresses a weakness in current two-party secure messaging tools: malicious participants can leak portions of a conversation alongside cryptographic proof of authorship, undermining confidentiality. The dissertation introduces two new authenticated key exchange protocols, DAKEZ and XZDH, with deniability properties that can prevent this type of attack when integrated into a secure messaging protocol. DAKEZ provides strong deniability in interactive settings such as instant messaging, while XZDH provides deniability for non-interactive settings such as mobile messaging. These protocols are accompanied by composable security proofs.

Secondly, the dissertation introduces Safehouse, a new protocol that can be used to implement secure group messaging tools for a wide range of applications. Safehouse solves the difficult cryptographic problems at the core of secure group messaging protocol design: it securely establishes and manages a shared encryption key for the group and ephemeral signing keys for the participants. These keys can be used to build chat rooms, team communication servers, video conferencing tools, and more. Safehouse enables a server to detect and reject protocol deviations, while still providing end-to-end encryption. This allows an honest server to completely prevent insider attacks launched by malicious participants. A malicious server can still perform a denial-of-service attack that renders the group unavailable or “forks” the group into subgroups that can never communicate again, but other attacks are prevented, even if the

server colludes with a malicious participant. In particular, an adversary controlling the server and one or more participants cannot cause honest participants' group states to diverge (even in subtle ways) without also permanently preventing them from communicating, nor can the adversary arrange to covertly remain in the group after all of the malicious participants under its control are removed from the group. Safehouse supports non-interactive communication, dynamic group membership, mass membership changes, an invitation system, and secure property storage, while offering a variety of configurable security properties including forward secrecy, post-compromise security, long-term identity authentication, strong deniability, and anonymity preservation. The dissertation includes a complete proof-of-concept implementation of Safehouse and a sample application with a graphical client. Two sub-protocols of independent interest are also introduced: a new cryptographic primitive that can encrypt multiple private keys to several sets of recipients in a publicly verifiable and repeatable manner, and a round-efficient interactive group key exchange protocol that can instantiate multiple shared key pairs with a configurable knowledge relationship.

Acknowledgments

The work presented in this dissertation would not have been possible without the encouragement, support, and assistance of many individuals. The few words in this section cannot adequately express my gratitude.

I would foremostly like to thank Ian Goldberg for his outstanding supervision during my MMath and Ph.D. programs at the University of Waterloo. Few students are afforded the privilege of working with a supervisor possessing such a remarkable ability and proclivity to establish foundations from which the students can pursue their goals, wherever those goals may take them. The results presented in this dissertation would simply not have been possible without his guidance, extensive knowledge of the field, and careful attention to detail. The research process is an arduous journey, and it was an invaluable comfort to know that any obstacle could be overcome given sufficient time and space on a whiteboard. I will be forever thankful to have benefited from his knowledge and wisdom.

Thank you to my coauthors—Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith—who worked with me to systematize secure messaging schemes, identifying the gap in the literature that ultimately became the focus of my research. Together, we were able transform an eclectic menagerie of techniques into structured advice that has driven extensive positive change in the subsequent years.

I would also like to thank the eponymous and anonymous reviewers of the publications that formed the basis of Parts I and II. Their insightful feedback and scrutiny challenged us to push further and do better. In particular, thank you to Trevor Perrin and Henry Corrigan-Gibbs for their helpful feedback regarding the publication underlying Chapter 2, to Markulf Kohlweiss for shepherding the publication that became Part II, to Peter Schwabe and David Jao for their comments that informed the material in Chapter 5, and to Alfredo Rial Duran for the exceptionally detailed verification of the security proofs in Chapter 6.

I would like to particularly give thanks to the members of the examining committee for this thesis—Nicholas Hopper, David Jao, Sergey Gorbunov, and Douglas Stebila—for graciously providing their time and insightful critiques. I am grateful for the committee members taking the time from their busy schedules to examine the thesis despite the significantly increased reviewing burden caused by the atypical length of this work.

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. The CrySP RIPPLE Facility significantly streamlined the parameter generation process described in Section 8.3.1 and the performance evaluation of Safehouse presented in Section 10.13, allowing the results to be collected in a reasonable amount of time.

During my time at the University of Waterloo, the Cryptography, Security, and Privacy (CrySP) research group has been far more than an institution. I will be eternally grateful for the friendships that I made in CrySP and the time that we spent together over these many years—the movie nights, the D&D sessions, the Plaza lunches, the conversational vortices, and all the rest. I have no doubt that the semantics and epistemology will continue, wherever we may meet again. I would especially like to thank the denizens of `#crysp` for the tether to humanity during the pandemic, and for invariably tolerating my rambling. One could not ask for more.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [funding reference numbers RGPIN-2017-03858 and STPGP-463324-14].

Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), [numéro de référence RGPIN-2017-03858 and STPGP-463324-14].

Natural Sciences and Engineering Research Council of Canada

Conseil de recherches en sciences naturelles et en génie du Canada

Canada

Dedication

To Garry and Theresa Unger

for their love and support,

to all who patiently tolerated my verbosity herein,

and to the health professionals of the world

who, granted far too little support and agency, selflessly confronted the horrors of the COVID-19 pandemic nonetheless.

Thank you.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Theorems, Definitions, & Propositions

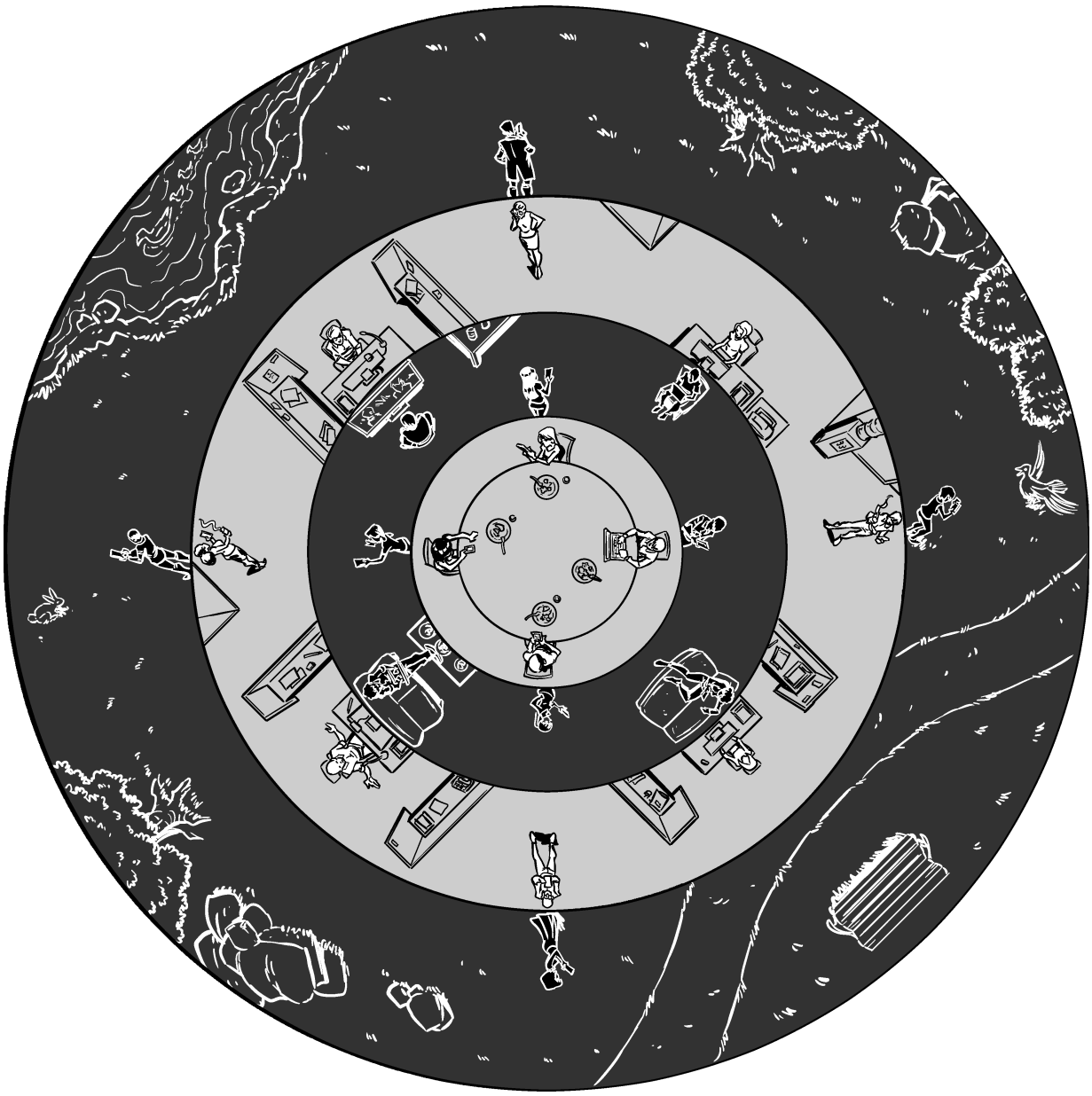
List of Schemes

List of Abbreviations

3DH Triple Diffie-Hellman (21 uses)
AEAD Authenticated Encryption with Associated Data (31 uses)
AKE Authenticated Key Exchange (24 uses)
ART Asynchronous Ratcheting Trees (25 uses)
BD Burmester-Desmedt (31 uses)
BDL Batch Discrete Logarithm (7 uses)
BDLEQ Batch Discrete Logarithm Equality (29 uses)
BGP Border Gateway Protocol (1 use)
BRAKEM Batch Recursive Attested Key Encapsulation Mechanism (210 uses)
CBRAKEM Contextual BRAKEM (55 uses)
CDH Computational Diffie-Hellman Problem (23 uses)
CG-DDLEQ Cross-Group Double Discrete Logarithm Equality (5 uses)
CG-DLEQ Cross-Group Discrete Logarithm Equality (12 uses)
CGKA Asynchronous Continuous Group Key Agreement (34 uses)
DAG Directed Acyclic Graph (6 uses)
DAKE Deniable Authenticated Key Exchange (120 uses)
DAKEZ Deniable Authenticated Key Exchange with Zero-knowledge (92 uses)
DDH Decisional Diffie-Hellman Problem (22 uses)
DDL Double Discrete Logarithm (35 uses)
DDLEQ Double Discrete Logarithm Equality (16 uses)
DEM Data Encapsulation Mechanism (3 uses)
DGKE Dynamic Group Key Exchange (25 uses)
DH Diffie-Hellman (138 uses)
DL Discrete Logarithm (55 uses)
DLEQ Discrete Logarithm Equality (28 uses)
DNS Domain Name System (1 use)
DRE Dual-Receiver Encryption (18 uses)
DREAD Dual-Receiver Encryption with Associated Data (31 uses)
ECDH Elliptic Curve Diffie-Hellman (4 uses)
ECPP Elliptic Curve Primality Proving (6 uses)
EC-PTEQ Elliptic Curve Plaintext Equality (8 uses)
EUC External-subroutine Universal Composability (17 uses)
GKE Group Key Exchange (93 uses)
GNFS General Number Field Sieve (2 uses)
GUC Generalized Universal Composability (50 uses)

IBE Identity-Based Encryption (10 uses)
IETF Internet Engineering Task Force (6 uses)
IND-CCA2 Indistinguishability under adaptive Chosen Ciphertext Attack (20 uses)
IND-CPA Indistinguishability under Chosen Plaintext Attack (16 uses)
IRC Internet Relay Chat (37 uses)
ITK Insider Secure TreeKEM (3 uses)
KC Key Control (75 uses)
KCI Key Compromise Impersonation (11 uses)
KDF Key Derivation Function (32 uses)
KEA Knowledge of Exponent Assumption (4 uses)
KEM Key Encapsulation Mechanism (52 uses)
KGM Key Graph Manipulation (22 uses)
KLL Kim-Lee-Lee (53 uses)
LVSM Label-Value Store Manipulation (48 uses)
MAC Message Authentication Code (16 uses)
MLS Message Layer Security (113 uses)
MPC Multi-Party Computation (7 uses)
NIKE Non-Interactive Key Exchange (5 uses)
NIST National Institute of Standards and Technology (7 uses)
NIZKPK Non-Interactive Zero-Knowledge Proof of Knowledge (216 uses)
OTR Off-The-Record messaging (23 uses)
PIR Private Information Retrieval (1 use)
PKG Private Key Generator (5 uses)
PKI Public Key Infrastructure (1 use)
PPT Probabilistic Polynomial Time (26 uses)
RFC Request For Comments (3 uses)
ROM Random Oracle Model (21 uses)
SBRAKEM Single BRAKEM (10 uses)
SNFS Special Number Field Sieve (6 uses)
SoK Signature of Knowledge (21 uses)
SVC Scalable Video Coding (4 uses)
TGDH Tree-based Group DH (14 uses)
TKLL Tree Kim-Lee-Lee (139 uses)
TLS Transport Layer Security (16 uses)
TTKEM Tainted TreeKEM (10 uses)
UC Universal Composability (15 uses)
X3DH Extended Triple Diffie-Hellman (22 uses)
XMPP Extensible Messaging and Presence Protocol (7 uses)

XOF Extendable Output Function (5 uses)
XZDH Extended Zero-knowledge Diffie-Hellman (61 uses)
ZDH Zero-knowledge Diffie-Hellman (98 uses)
zk-SNARK Zero-Knowledge Succinct Non-interactive Argument of Knowledge (55 uses)
zk-STARK Zero-Knowledge Succinct Transparent Argument of Knowledge (11 uses)



CONIUGERE SIT SECLUDERE

To join together may be to cut off.

CHAPTER 1 | Introduction

Our society has become heavily dependent on electronic communication, with our most critical discourse now being conducted over the Internet and other digital communication networks. Securing and protecting the privacy of modern digital communication tools has never been more important. At the same time, public awareness of data security and privacy has increased in response to revelations of government surveillance, high-profile data breaches, negative consequences of concentrated corporate control over communications, and other incidents. Shortly after the Snowden revelations, the American public expressed an interest in gaining control over information about them stored online [Mad15]. Since then, worldwide concern about online privacy has increased, and a majority of users believe that social media, search engine, and internet technology companies hold too much power [Bri18]. Similarly, most Canadians are worried about the privacy and security of their personal information, and four in ten have altered their behavior due to data privacy concerns [Sim18]. A majority of Americans continue to feel that companies and governments are collecting too much of their personal information, that their data has become less secure, and that they have little control over this collection [ARA+19]. At the same time, the privacy situation has become worse: widespread quarantines issued to slow the COVID-19 pandemic have caused mass adoption of communication tools with woefully inadequate security and privacy properties, such as the Zoom group videoconferencing application [BBG+20; Hod20]. Due to this widespread concern about data security and privacy, mass adoption of inadequate tools, increased activity by adversarial actors, and a general sense that nothing can be done to improve the situation, there is an opportunity for change. An important part of that change is a technological improvement to existing tools.

As a technical tool to rearrange power relationships [Rog15], cryptography can help to address some of the security and privacy problems with digital communication technologies. In particular, end-to-end encrypted secure messaging protocols can mitigate some of the threats to user privacy. By cryptographically restricting the platform's access to communication data, systems can continue to reach Internet scale while eliminating the most egregious adversarial attacks, such as mass surveillance. Ever since the Snowden revelations began in 2013, there has been an explosion of secure messaging protocols, components, and tools that attempt to accomplish this goal [UDB+15]. Following adoption by several major platforms [Mar14a; Mar16;

Lun18], the Signal protocol [Ope13] has become the leading standard for achieving strong security in the "two-party setting" (i.e., where two users exchange secure messages with each other). Noteworthy protocols and applications that compete against Signal include: the Off-The-Record messaging (OTR) protocol [BGB04], which is used by many applications to secure communications [OTR14]; Wire [WIRE15], a web-based messenger and protocol incorporating techniques from both Signal and OTR; and Ricochet [Ric14], a messenger with a focus on hiding conversation metadata. There are many other secure messaging applications that have gained popularity without academically remarkable protocol features, such as Telegram [Tel14] and Briar [RST+14].

While all of these applications and protocols offer attractive security features for two-party text messaging, there is a generally a lack of support for encrypted conversations among groups of three or more users, also known as "secure group messaging". This omission is understandable due to the complexity of the problem, but it is also very significant: user studies show that secure group messaging is one of the most important features for users [EHM17; HEM18]. Applications that *do* offer secure group messaging to the user, such as Signal, almost always do so using a two-party protocol to secure pairwise messaging channels. For example, the Signal protocol for group text messaging involves establishing pairwise Signal conversations. The client uses hybrid encryption to protect the messages, where the Key Encapsulation Mechanism (KEM) takes place over the pairwise channels, and the Data Encapsulation Mechanism (DEM) is broadcast to all members of the group by a central server that duplicates the ciphertext. While this approach provides basic security properties such as confidentiality and authentication, its computation and communication costs scale linearly with the group size due to the pairwise KEM operations, and the technique makes certain security properties prohibitively expensive [CCG+18]. These limitations exclude the possibility of securing important modern applications such as virtual meetings, institutional communications, and large communities such as those supported by Slack and Discord. A better approach is warranted.

Unfortunately, while the two-party case has been researched extensively, protocols for secure group messaging are only just beginning to emerge. Prior academic publications that propose complete secure group messaging systems [KTN04; RKAG07; GUVC09; LVH13; HLZZ15; SVH18; SH19] are not suitable for modern group messaging scenarios of interest due to a lack of scalability, missing features (such as *asynchronicity*—the ability to be used on intermittently connected mobile devices), or the lack of critical security properties. Partly due to the lack of a complete and robust academic foundation for secure group messaging, several public protocol development efforts emerged around 2015–2016, including (n+1)sec [eQu15], flute [Kad16], and mpEnc [LK16]. Each of these initiatives lacked important features and/or security properties. Eventually, the community consolidated its efforts into the Message Layer Security (MLS) protocol [BBM+20], forming an Internet Engineering Task Force (IETF) working group that has

drafted the most advanced secure group messaging solution currently available. Unlike prior academic and public efforts, MLS provides a vast set of security properties, is widely applicable to secure communication scenarios, and has existing proof-of-concept implementations. However, it is missing one important security property: *insider security*.

A protocol with insider security is one that prevents malicious participants (e.g., users in a group conversation) from attacking the protocol. The exact meaning of "insider security" is thus dependent on the protocol's security goals, so it is more accurately described as a family of related security properties. One aspect of insider security for secure messaging is preventing participants from provably leaking conversation plaintexts—either maliciously or under coercion. In the context of a secure group messaging protocol, insider security further ensures that all participants have a consistent view of the set of participants and messages (where the exact meaning of "consistent" may be strict or loose depending on the purpose of the protocol). For example, when naïvely building a secure group messaging protocol from pairwise two-party secure messaging channels, a malicious server can inject message recipients for a subset of legitimate users, or selectively deliver messages to only a few users. Real-world deployments of Signal have proven to be vulnerable to such attacks [SKH17; RMS18]. MLS is designed to store all group state at the endpoints rather than in routing infrastructure. While this choice results in excellent scalability properties, it comes with several disadvantages: there is no authoritative source for the "true" group state, spam prevention is frustrated, and insider security is damaged [ACJM20]. For example, the MLS draft specification points out how a single malicious insider can cause the entire group to fragment into disjoint subgroups with a denial-of-service attack [BBM+20, §10.2]. Insider security becomes more important as groups become larger because there are more potential attack vectors. Since protocols like MLS are designed to scale to support large groups, insider security is more important than ever before.

This dissertation improves the robustness of modern end-to-end secure messaging protocols by introducing efficient new key exchange protocols to prevent attacks by malicious insiders while conservatively relying on only basic and widely accepted security assumptions. Specifically, the new DAKEZ, ZDH, and XZDH protocols prevent insiders from provably leaking messages in two-party secure messaging protocols, even when they interactively collaborate with the party attempting to verify the leaks. XZDH is designed to be a drop-in replacement for the existing mechanism used by the Signal protocol. The dissertation also introduces Safehouse: a comprehensive solution for secure group messaging in a wide range of scenarios that enforces correct participant behavior, thereby preventing insider attacks. All of these new protocols support "non-interactive" or "asynchronous" scenarios where all secure message recipients are temporarily offline when a message is sent; this feature is critical for mobile use cases where network connectivity is transient and devices may be offline for extended periods of time.

1.1 Thesis Statement

This dissertation establishes the following:

> We can design efficient key exchange protocols that provide strong end-to-end security properties for low- and high-risk users in common secure messaging scenarios, such as instant messaging, mobile messaging, team communication, and group video conferencing, while relying on only common security assumptions. In particular, it is possible to protect against attacks by malicious insiders and coerced users even while operating in an asynchronous network setting where devices may be disconnected for long periods of time without warning.

1.2 Contributions

At the core of this dissertation are the new cryptographic protocols that provide insider security. However, it also includes many other academic contributions of independent interest. The most notable contributions are:

1. **A survey of group key exchange protocols.** Chapter 3 surveys and categorizes prior approaches to group key exchange protocols, which are an important building block for secure group messaging systems.

2. **A new interactive deniable authenticated key exchange protocol with offline and online deniability.** Section 5.4 presents DAKEZ, a new protocol for securely establishing a shared key between two parties that can interactively communicate. DAKEZ prevents an insider (i.e., one of the two alleged participants) from provably sharing the key with an outside party. Consequently, leaked conversations are indistinguishable from pure forgeries and alleged leaks can never be cryptographically authenticated.

3. **A new non-interactive deniable authenticated key exchange protocol with offline deniability, partial online deniability, and configurable forward secrecy.** Section 5.6 presents XZDH, another new key exchange protocol. Unlike DAKEZ, XZDH is a drop-in replacement for Signal's key exchange protocol. It prevents insiders from provably sharing the key with an outside party in most, but not all, situations. This is a strict improvement compared to Signal's

existing guarantees, as XZDH preserves all of Signal's other desired security properties. Unlike DAKEZ, XZDH can be used to send secure messages to a recipient that is temporarily offline.

4. **Composable proofs of security for the aforementioned protocols.** Chapter 6 provides comprehensive proofs that the new two-party key exchange protocols in this dissertation satisfy all of their promised security properties. These are simulation-based proofs in the universal composability framework, which provides extremely powerful composability guarantees. Specifically, this allows the protocols to be incorporated into larger secure messaging protocols in a safe manner without the need to re-prove the underlying security properties. The security proofs include novel definitions of idealized protocol functionalities that are applicable to a broad class of key exchange schemes.

5. **A new cryptographic primitive: batch recursive attested key encapsulation.** Chapter 8 defines a new cryptographic primitive that allows a prover to encapsulate a set of private keys to sets of receivers in a publicly verifiable manner. Moreover, the private key being encapsulated corresponds to the same group as the recipients' keys, so the entire process can be repeated with the encapsulated key as a new receiver. This primitive has widespread applications, but this dissertation focuses on the application to secure group messaging.

6. **A new round-efficient interactive group key exchange protocol for populating key trees.** Chapter 9 presents a new protocol that allows an interactively communicating set of participants to establish a set of shared keys according to a pre-determined access policy using exactly four communication rounds. This protocol is publicly verifiable and secure against active network attackers. It is generally applicable to any protocol that requires establishing a hierarchical set of shared keys. The protocol can be used to accelerate the process of adding users to a secure group conversation.

7. **A new comprehensive secure group messaging protocol with insider security.** Chapter 10 presents the Safehouse protocol. Safehouse takes responsibility for managing all of the shared cryptographic keys involved in a secure group messaging system and handling invitations to join the group. Developers can implement secure group messaging tools that use Safehouse for key management, avoiding the need for custom application-specific designs and automatically inheriting Safehouse's security guarantees. Safehouse grants the developer complete control over how the shared group key is used to encrypt and transfer messages, making it suitable for a wide variety of applications. Safehouse requires the presence of a server that is responsible for ordering, storing, delivering, and verifying the correctness of messages. MLS recommends such a server, but does not require it [BBM+20, §12.1]. In contrast, Safehouse provides cryptographic tools that can be used by the server to reject spam and enforce group access control policies. A misbehaving server can attack the availability of the group, but nothing

else. Chapter 11 implements a complete group chat system using Safehouse that approximates the user experience of IRC using a server-side "bouncer" proxy, including a complete GUI client and a server backed by MySQL. This proof-of-concept deployment demonstrates the viability of the Safehouse protocol for one of its intended applications.

1.3 Thesis Organization

The new cryptographic protocols presented in this dissertation are very complex. Moreover, each protocol is examined from a variety of perspectives that span a broader range of disciplines than one might typically expect to find in a single document. Within this dissertation, the reader will find a mix of abstract cryptographic definitions, discussions of design concerns, very low-level implementation details, textbook-like background information for developers interested in implementing the protocols, solutions to problems that arise only during practical deployments, empirical performance evaluations, and mathematically intensive security proofs. This thorough treatment of the subject matter means that this dissertation is atypically long, but it also means that most readers will be interested in only a subset of the content types.

To help readers quickly identify content of interest, this document uses symbology to annotate chapter and section headings with the type of content that they contain. Each section heading includes symbols indicating the type of content in the section (and any subsections within it). Each chapter begins by listing the union of all content types appearing within its sections in order of first appearance, as well as a textual reminder of the meaning of the symbols. These annotations allow readers to quickly identify if a chapter or section contains relevant content of interest. Table 1.1 lists the symbols that are used throughout the document.

Figure 1.1 depicts the relationship between chapters in terms of expected prerequisite knowledge. It also provides shorthand titles, lengths, and content types for each chapter. When skipping to a particular chapter, readers may find it helpful to refer to this diagram in order to quickly identify the assumed knowledge.

Table 1.1 CHAPTER AND SECTION CONTENT TYPE SYMBOLOGY. (Ref: 6)

Annotation	Meaning
Background	Sections with this annotation contain background information such as surveys of related work, notation, and definitions. These sections can help to understand the protocols, but they can also be treated as reference material.
Discussion	Sections with this annotation discuss aspects of the core contributions in terms of a broader context. In particular, these sections discuss motivation for design choices or practical deployment concerns for the protocols.
Protocol	Sections with this annotation contain the core contributions of the dissertation. These sections introduce the new protocols and the supporting material required to define them.
Implementation	Sections with this annotation contain implementation details, either as general advice, or concerning a specific implementation that was developed as part of this work. These sections are primarily of interest to developers.
Evaluation	Sections with this annotation contain empirical performance evaluations, providing insight into the practicality of the protocols in various configurations.
Proof	Sections with this annotation contain security proofs or constructions that are only necessary for the purpose of providing a security proof. These sections are primarily of interest to cryptographers and readers wishing to verify the claimed security properties.

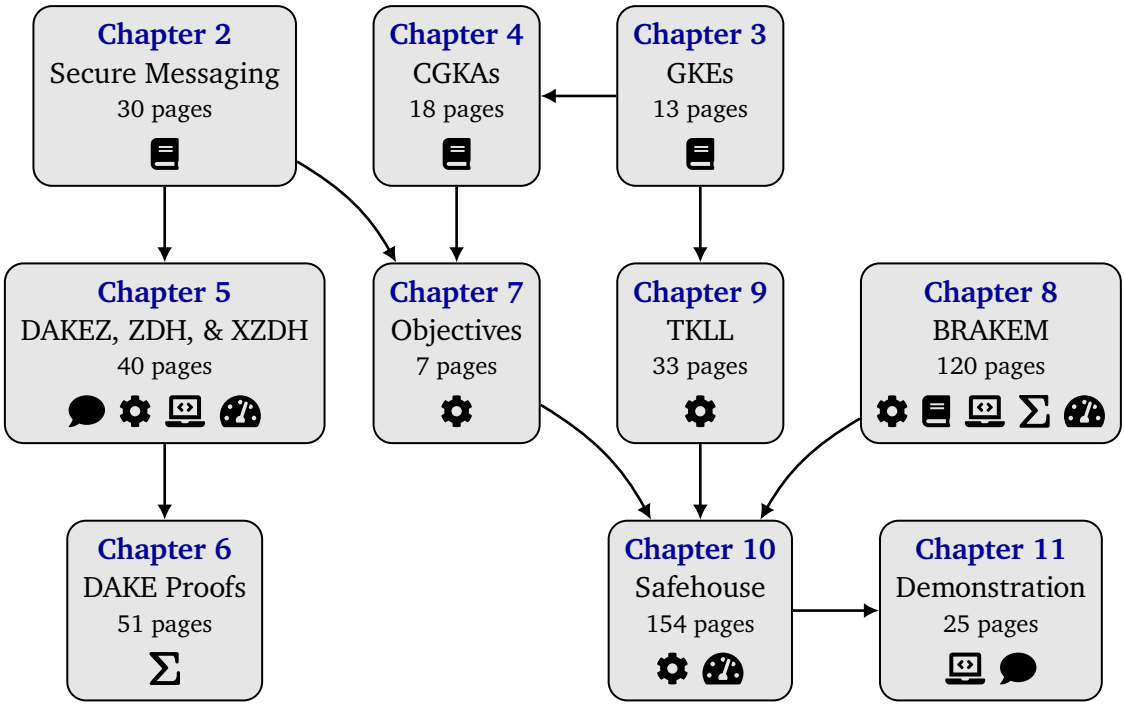


Figure 1.1 CHAPTER DEPENDENCIES AND CONTENT TYPES. (Ref: 6)

PART I

Background

CHAPTER 2 | Understanding Secure Messaging

In this chapter: Background

ECURE messaging systems vary widely in their goals and corresponding design decisions. Security properties that are desirable to one group of users may be undesirable to another group. Additionally, their target audiences often influence how they are defined. A protocol developed by a working group of developers might be specified solely as network message structures, whereas a protocol defined in a cryptography journal article might be specified solely in terms of abstract algebra. It is important to understand the history of secure messaging protocol designs and how they are presented in order to build and evaluate protocols that overcome new challenges. This chapter defines terminology to differentiate these designs and provide a foundation for an exploration of the secure messaging design space.

Portions of this chapter have been adapted from previous work by Unger et al. [UDB+15], a collaborative publication in which the author systematized secure messaging schemes. Interested readers will find additional details in the publication. The adapted portions have been significantly updated to cover important developments in the intervening years. The material concerning deniability in Section 2.4 and Section 2.5 is partially adapted from previously published work by Unger and Goldberg [UG15; Ung15].

2.1 Usability Studies

The incentives for developing cryptographic protocols like secure messaging schemes are often at odds with the needs of real users. For academic publications, the path of least resistance is often developing a novel cryptographic technique first, and then later finding a secure messaging scenario to serve as "motivation". For software developers, the incentives are to quickly produce a differentiated product with a compelling story. It is usually not worth the effort to scan the academic literature for advanced cryptography that may be beneficial, but would require significant development resources to implement at a production-ready level of quality.

The best outcomes for users are achieved when we take a step back, investigate the scenarios in which a solution can benefit users, derive a precise set of requirements, develop cryptographic tools to build an appropriate protocol, and produce a proof-of-concept implementation that can be used as a guide and reference for production-ready libraries. This is a time-consuming process that is difficult to get right. Luckily, we can learn a lot about the problems that need to be solved by consulting the results of user studies. These studies provide insights into the threat models of users, the security and privacy properties that they expect, and how they value these properties in the context of their overall relationship to technology. Considering the results of these studies is important because even the most theoretically secure protocol does not provide any security in practice if nobody uses it.

This section surveys user studies that have found usability problems and other factors limiting adoption of secure messaging tools.

2.1.1 Encrypted Email Usability

Although encrypted email is not normally considered to be part of "secure messaging" (a term that refers primarily to instant and mobile messaging), studies of the usability of secure email tools spans decades. Many of the results from these studies are applicable to the secure messaging domain, and so it is worthwhile to consider them. The seminal paper on usable secure email is due to Whitten and Tygar [WT99]. In "Why Johnny Can't Encrypt", the authors investigate the usability of PGP 5.0 using cognitive walkthroughs [WRLP94] and a user study. Their findings were shocking: the majority of participants could not encrypt and sign an email within 90 minutes, and three of twelve participants accidentally sent unencrypted emails. They concluded that "effective security" requires distinct user interface design techniques. This paper sparked a long line of follow-up studies that are colloquially known as "Johnny" papers in reference to the recurring character in the publication titles.

Garfinkel and Miller [GM05] conducted a user study similar Whitten and Tygar's study, but with a focus on S/MIME in Outlook Express. Unlike PGP, which requires users to manually verify the correctness of public key fingerprints, S/MIME verifies identities using certificate authorities and automated certificate pinning. The authors argue that the poor usability uncovered by Whitten and Tygar was primarily caused by the complex fingerprint verification mechanism in PGP, rather than by the software's user interface. They found that automated key management dramatically improves usability, but that participants still had difficulty with making trust decisions and determining who can read encrypted messages. Sheng et al. [SBKH06] repeated Whitten and Tygar's study (seven years later) and found that users were still overwhelmingly unable to complete the tasks, despite years of interface improvements to PGP. They also found that

users were troubled by software that did not show the ciphertext, because it was unclear if the encryption software was working. During the same year (2006), Gaw et al. [GFF06] investigated user opinions of encrypted email within an activist organization. They found that users thought email encryption was "justified" for secret communications, but that it was "paranoid" to encrypt day-to-day communications.

Clark et al. [CGM+11] investigated the use of an encrypted two-way radio technology used by federal law enforcement surveillance operatives in the USA. They were able to intercept hundreds of hours of accidentally unencrypted sensitive communications over two years. They found that these errors were primarily caused due to the ability of users to switch radios into an unencrypted transmission mode, the inability to see the mode of received transmissions, and a centralized and inflexible group keying architecture. Fahl et al. [FHM+12] investigated user preferences for encrypted messaging on Facebook. Their results revealed a strong preference for automated key management. Users also normally opted for automated encryption, but did not express a preference for it. A usability analysis by Moecke and Volkamer [TV13] led to a set of criteria for usable end-to-end encryption tools: it should be easy to join the encrypted communication network, the provided tools should provide the information necessary to make informed trust decisions, there should be no requirement to understand public key cryptography, and secure communication should be permitted without a requirement for out-of-band identity verification.

Renaud et al. [RVR14] conducted an exploratory qualitative study investigating barriers to adoption of end-to-end encrypted email. Their results suggest that most users never reach a stage where the poor usability of tools is the problem. To reach that point, users must first have an awareness of privacy violations, be concerned about privacy, achieve a full understanding of the issues, see a need to act, and know that end-to-end encryption can help [RVR14]. Their conclusion is that researchers and developers should focus on building comprehensive end-user mental models for secure email through education.

Several studies have considered usability concerns surrounding automatic email encryption. Ruoti et al. [RKB+13] studied user interactions with Pwm (pronounced "poem"), a transparent (i.e., non-visible) automatic encryption overlay for Gmail that uses key escrow. They compared the usability of Pwm with a mockup called "Message Protector" (MP), a standalone desktop application where users manually encrypt and decrypt texts. Somewhat surprisingly, they found that users trusted MP more than Pwm and gave both comparable usability ratings, but users expressed a preference for tight browser integration. Their conclusion, similar to the much earlier hint from Sheng et al. [SBKH06], was that some encryption details should be exposed to the user as part of a tightly integrated system. Atwater et al. [ABH+15] suspected that this trust disparity was due to confounding factors like user interface polish and inherent trust differences between browser plugins and desktop applications. They designed three types of email encryption tools to test this hypothesis, among others. They found that users strongly prefer encryption

systems to be integrated in email clients, visibility of ciphertext itself does not affect user trust, the main source of trust is the developer's online reputation, and showing ciphertext does not improve usability beyond having clear indications that the message is encrypted. Their work provides three key design goals for secure messaging systems: setup should be easy, basic tasks should be accomplishable without adjusting settings, and the effect of buttons and the encrypted status of messages should be clear and explicit. Ruoti et al. [RAH+16b] conducted a follow-up study with an updated version of Pwm that found similar effects. They found that automatic encryption does not have any usability impact when appropriate user interface indicators are present. These findings were confirmed yet again in additional studies where pairs of participants were tasked with sending messages to each other, rather than to the researchers [RAZS15; RAH+16a; RAD+19].

2.1.2 Secure Messaging Usability

Historically, usability studies in secure communication have focused on encrypted email. The rise of secure messaging applications in recent years has caused an increased focus on the instant and mobile messaging domains: dozens of usability studies specific to secure messaging have been published since 2014. This section surveys the most notable results.

Several studies have identified that users adopt secure messaging tools for reasons that may be unexpected. Roesner et al. [RGK14] examined user perceptions and behaviors surrounding the "self-destructing messages" feature of the popular photo sharing tool Snapchat. Contrary to what one might expect, they found that users typically do not send sensitive content through Snapchat (although 25% claimed they had done so "experimentally"). Moreover, users were generally aware that the security guarantee of a "self-destructing message" is theoretically impossible due to, among other things, the ability to use a second device to capture the screen (an example of the "analog hole"). Most understood that "deleted" messages could be recovered, and the practice of screenshotting photos was common and expected. Most users decided to use Snapchat for reasons unrelated to security or privacy. De Luca et al. [DDO+16] found similar user motivation more broadly: peer influence, not security or privacy, primarily drives the adoption of messaging tools with advertised security features. Elliott and Brody [EB16] found that vulnerable users in New York appreciated Snapchat's "self-destructing message" policy because "shoulder surfing" and physical device compromise were significant and constant threats. Geeng et al. [GHR20] surveyed "sexting" (i.e., exchanging messages containing sexually explicit photos) practices across multiple messaging platforms. They recommended that platforms supporting this use case should provide a mechanism to manage unsolicited sexts, support a "self-destructing" messages feature, but also normalize and secure the process of overriding the "self-destruction" in order to save the content of a message.

Usability concerns are not the main barrier to adoption of secure messaging tools, and when the tools *are* adopted, users often lack correct mental models of what the tools do or how to safely use them. Abu-Salma et al. [ASB+17] performed a comprehensive analysis of barriers to the adoption of secure messaging tools and found results consistent with the work of Renaud et al. [RVR14]: lack of understanding, concern, and accurate mental models are more common obstacles than poor tool usability. One of the primary obstacles for niche security-focused tools is the lack of interoperability and fragmented user bases. As some participants pointed out, a messaging tool is useless if it cannot send messages to any of a user's contacts. Low quality of service (QoS), such as poor audio/video connections or poor interface design, is another significant obstacle: users perceived tools with lower QoS to have lower security. This finding echoes the earlier results from Atwater et al. [ABH+15]. Abu-Salma et al. [ASB+17] also found that the sensitivity of messages does not drive the adoption of secure messaging tools; users consider these tools to be primarily for "informal" communication, while sensitive information should be sent through "formal" channels, or split across multiple channels. Users also expressed inaccurate mental models of communications technology and security: users typically think of each application as a separate "channel" (rather than adopting a network-centric view), believe that security requires obscurity, and misunderstand adversarial capabilities. Consequently, users described insecure behaviors and practices: they ranked the security of communication channels incorrectly (voice calls and SMS messages were considered more secure than secure messaging tools), considered openness of protocol designs and audits to be *negative* security properties, believed that service providers could read end-to-end encrypted content, and expected to be informed when governments or operators accessed their account records. Users also generally viewed attempts to secure communications as futile. As a result of this study, Abu-Salma et al. recommended that researchers focus on improving existing messaging tools with proven utility instead of developing niche secure messaging applications, work to improve the QoS of tools, and understand the target population and their mental models. These findings continued to hold true in subsequent studies. Abu-Salma et al. [AKP+17] and Vaziripour et al. [VFO+18] found that Telegram [Tel14] users had an indistinct feeling of security (likely due to the QoS and marketing) despite consistently failing to activate the opt-in end-to-end encryption feature. Gerber et al. [GZH+18], Abu-Salma et al. [ARUW18], and Dechand et al. [DNDS19] all confirmed that most users believe SMS to be more secure than secure messaging tools, believe that end-to-end encryption is easily defeated (e.g., by skilled neighbors or service providers), and switch to insecure channels to send sensitive content.

Some studies have identified a disconnect between users' mental models of secure messaging technology and signaling used by the tools. Wu and Zappala [WZ18] examined user mental models of encryption and found four categories of models with various levels of correctness. All users conceptualized encryption in terms of symmetric keys. The users in the study considered en-

cryption to be a tool reserved for illicit, immoral, or paranoid activities. Demjaha et al. [DSB+18] extracted mental models of end-to-end encryption from study participants, generated metaphors from these models, and reflected them back to other participants as an educational tool. They found that metaphors generated in this manner were harmful: they undid correct understanding that users already had and did not improve understanding of end-to-end encryption. However, they also found that these metaphors were *less harmful* than metaphors already used in practice that were not derived from user mental models. They conclude that explanatory metaphors for end-to-end encryption should be avoided.

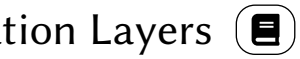
Ermoshina et al. [EHM17] examined secure messaging tools from a different angle: they compared the needs of users to the perceived needs of users according to developers. Moreover, they differentiated between "low-risk" and "high-risk" users based on the potential for imminent concrete physical harms due to failures in information security, and criticized previous researchers for focusing exclusively on low-risk users (typically students in the United States and Western Europe). They expanded these results in a subsequent follow-up study [HEM18]. They found that the needs of low-risk and high-risk users differ, and that developers often focus on security properties that both user categories report as unimportant. The two groups differ in terms of whether they are primarily concerned about *passive* adversaries (adversaries that monitor and capture network traffic but do not modify traffic or participate in protocols) or *active* adversaries (adversaries that may create, alter, or delete network traffic, or directly participate in protocols). Low-risk users are most concerned about global passive adversaries (i.e., passive adversaries with a global view of the network) and server seizures. They are indifferent about metadata collection (possibly due to unfamiliarity with the power and impact of the practice). Of all of the secure messaging tool features discussed in the study, low-risk users rated only support for group conversations as "high importance". In contrast, high-risk users are most concerned about local active adversaries capable of physical device compromise and other targeted attacks. These users rate both group conversations and metadata protection as "high importance". They prefer open-source systems, but without an informed rationale. The high-risk users consider key verification to be important, but prefer to verify identities out-of-band instead of using cryptographic key verification protocols. Finally, while they are concerned about metadata leaks, their meaning of the term differs from that of developers: they are primarily concerned about distribution of metadata (e.g., phone numbers and account associations) to their contacts, rather than network characteristics observed by a government surveillance entity. In particular, high-risk users considered pseudonyms to be a very important feature. Neither high-risk nor low-risk users cared about deniability, decentralization, or openness in protocol design. Dev et al. [DMC20] examined the similarities and differences between privacy concerns in Saudi Arabia and India. Their study resulted in three key recommendations: avoid automatically sharing contact information that could be used to contact users outside of the secure messaging platform

without their consent, require users to explicitly consent to being added to a group chat, and enable information sharing boundaries based on group types (e.g., professional versus personal groups).

In order to evaluate the impact of human behavior on security protocols, Ellison [Ell07] introduced the concept of a *ceremony*. In addition to the standard machines and network connections in most security models, the ceremony also contains human "nodes", connections between the humans and the machines (e.g., through user interfaces), and connections between the humans. The ceremony captures all of the behavior that is normally described as "out-of-band". Given empirical measurements of human behavior during the ceremony, this framework can be used to assess the real-world security of a protocol that involves human interaction. Vaziripour et al. [VWO+18] used this framework to evaluate the *authentication ceremony* in the Signal [Ope13] app—the authentication ceremony involves having the users compare "safety numbers" out of band in order to defend against active network attackers.[1] Users were not aware of the existence of the ceremony, had difficulty completing it when prompted, and were confused about the purpose even after completing it. The authors evaluated an alternative approach using opinionated design: by explicitly prompting users and guiding them through the ceremony (and allowing remote authentication based on vocal recognition), the success rate increased drastically, but users still did not understand the purpose of authentication. In stark contrast to this approach, several studies have called for secure messaging applications to be redesigned around users' existing mental models and values, rather than prescribing behaviors as in opinionated design. Dodier-Lazaro et al. [DABS17] evaluated several areas of security research, including secure messaging, in terms of "value-sensitive design". Under this approach, users' engagement and adherence to the designer's security goals is mediated by the users' values. Modeling these values helps to understand failure cases and to design viable alternatives. In the case of secure messaging tools, they point out that users are primarily driven by the desire to communicate with their peers as conveniently as possible, rather than security goals. Fassl [Fas18] subsequently used value-sensitive design to examine users' mental models and evaluate three alternative authentication ceremonies derived from these models: combination locks for the conversation, pseudo-passport IDs to compare, and live photo prompts similar to those used by Reddit users to "prove" their identity. These ceremonies all have serious disadvantages, but illustrate the general user-centric design approach. Wu et al. [WGH+19] used value-sensitive design to re-imagine the authentication ceremony in Signal. They evaluated user interpretation of various icons and phrases, then framed the ceremony as a "privacy check". In a user study where participants were

[1] ^ The "safety numbers" are derived from a cryptographic hash of a key exchange transcript. If an active network attacker modifies any of the transmitted key material, the numbers will not match. When two users confirm that the safety number matches between their devices, future communications between them are protected. This concept is borrowed in Section 11.2.3 as part of the Safehouse demonstration.

tasked with sending non-sensitive messages through the redesigned Signal, they found that users were consciously choosing to skip the authentication ceremony as an informed trade-off. The main result of the work is a demonstration that risk communication can empower users to make security decisions aligned with their values.

2.2 Specification Layers 

When it is time to design and specify a secure messaging protocol, one must first decide the level of abstraction of the specification. There are three broad levels of abstraction that are normally used to describe a secure messaging protocol:

1. **Abstract protocols**: At the most abstract level, protocols can be defined as sequences of values exchanged between participants. This mode of specification deals with high-level data flows and often omits details as significant as the choice of cryptographic protocols (e.g., key exchanges) and primitives to use. For example, an abstract protocol for two participants A and B might specify that "A sends to B a value $H(m)$ where H denotes a cryptographic hash function applied to the message m"; note that this description omits the mechanism of transmission, the method of encoding the hash output, the definition of the hash function, the encoding of the message input to the hash, and other implementation details. Academic publications typically specify protocols this way.

2. **Wire protocols**: Complete wire protocols aim to specify a binary-level representation of message formats. A wire protocol should be complete enough that multiple parties can implement it separately and interoperate successfully. Wire protocols often contain versioning mechanisms to ensure compatibility as changes are made over time. Implicitly, a wire protocol implements some higher-level abstract protocol, though extracting it may be non-trivial. Internet standards such as Requests For Comments (RFCs) released by the IETF typically specify protocols this way.

3. **Tools**: Tools are concrete software implementations that can be used for secure messaging. Tools may take the form of an end-user application, or as a software library that fully implements a secure messaging interface and can be embedded in an end-user application. Implicitly, a tool contains a wire protocol, though it may be difficult and error-prone to derive it, even from an open-source tool.

In practice, each type of specification provides distinct benefits. It is easiest to analyze the ideal security properties of abstract protocols and these are most likely to receive effective outside

review from experts as they can be read and understood quickly. Specifying abstract protocols encourages reuse of technology and derivation of new designs. Specifying wire protocols allows for the development of interoperable tools with differing interfaces or intended use cases and can surface practical problems often glossed over by abstract protocols (such as message padding or concatenation bugs). Wire protocols are most often read by software developers and are best used to specify systems that are intended to be implemented by people other than the specification authors. This separation of authorship is useful for federated systems and to provide end users with choice about which implementation to trust. Finally, the availability of tools, particularly with open-source code that facilitates code auditing, is critical for the adoption of protocols by end users. Moreover, developing a tool enables the collection of real-world performance data. Analyzing performance data is important because modern computing hardware is extremely complex and it is difficult to accurately predict the performance of algorithms for abstract or wire protocols. Discovering the location of performance bottlenecks in tools can often lead to protocol refinements that affect all specification layers.

Very few secure messaging systems define up-to-date abstract and wire protocols while also making a tool available, with a notable exception being OTR [BGB04; OTR16; OTR14].

2.3 Threat Model

The usability studies discussed in Section 2.1.2 primarily focused on scenarios where endpoints do not share any affiliation. However, "messaging" is a very broad objective that covers a wide range of applications. In many scenarios, all of the endpoints trust the same third party (e.g., a collection of microservices might trust a company's key distribution system, or a set of users might all trust the same software provider). In those scenarios, there is no need for sophisticated cryptography: the endpoints can simply establish secure channels with the trusted party and use a traditional messaging protocol, such as Internet Relay Chat (IRC) over Transport Layer Security (TLS), or an on-premises Slack-like service such as Mattermost [Mat15]. Thus, when building a secure messaging protocol (or more generally, any security system) it is important to precisely define the scenario(s) of interest and the associated threat model. The threat model specifies what information the adversary can obtain, the actions that it can perform, and its computational capabilities.

This dissertation focuses on practical systems that protect low- and high-risk users who do not trust a shared third party to protect message confidentiality or integrity. The adversary in this threat model is bound by real-world computational constraints. Notably, the adversary is bound to classical computation at realistic time scales and is not permitted to run quantum

algorithms. Additional research is required in order to fully secure the new protocols against quantum adversaries.[2] The logical adversary represents a potential group of colluding attackers. These attackers may include:

- **Local Adversary (active/passive)**: An attacker controlling local networks (e.g., owners of open wireless access points). An active attacker can manipulate the traffic sent across the network, whereas a passive attacker only monitors and records the legitimate traffic.
- **Global Adversary (active/passive)**: An attacker controlling large segments of the Internet, such as powerful nation states or large Internet service providers.
- **Service providers**: For messaging systems that require centralized infrastructure, the service operators are considered to be potential adversaries.
- **Insiders**: Parties to the conversation that are under some degree of adversarial control, either because they are legitimate parties that are being coerced, or because they are entities created by the adversary.

Any entity that is not party to the secure conversation must not be trusted to protect any security property aside from availability.[3]

The exact capabilities of the adversary when acting in the aforementioned roles is affected by the security property under consideration. For example, the games for some security properties (e.g., confidentiality) will assume that adversarial corruptions are "all-or-nothing", whereas in other scenarios (e.g., for post-compromise security) the extent of adversarial corruption might be limited (e.g., able to read ephemeral secrets, but unable to sign messages with long-term secrets). Even in cases where the adversary does not control an insider in a particular conversation, a strong threat model for secure messaging must assume that the adversary is enrolled in the messaging system. In particular, the adversary must always be able to start conversations, send messages, or perform other actions afforded to an ordinary user.

[2] ^ The new key exchanges presented in Part II include a mechanism to defend against future quantum adversaries. Defending against current quantum adversaries would require the use of quantum-resistant ring signatures. The Safehouse protocol presented in Part III employs many primitives for which quantum-resistant equivalents are unavailable. Most significantly, a quantum-resistant version of Safehouse would require a quantum-resistant instantiation of the new cryptographic primitive defined in Chapter 8. This is a major challenge that is left to future work.

[3] ^ In practice, there are usually many different entities that can attack the availability of a protocol that operates over the Internet, such as Border Gateway Protocol (BGP) participants, Domain Name System (DNS) servers, and routers. Security techniques that provide greater availability, such as anonymization (to defend against targeted attacks) and replication, are largely orthogonal. Protocols that introduce new points of failure (e.g., a central server) can often have these weaknesses mitigated by making performance trade-offs (e.g., replacing a server with a computing cluster that runs protocols to emulate a single server).

2.4 Deniability

Deniability is a particularly interesting security property to consider in the context of secure messaging. In the original publication of OTR, Borisov et al. argued that unrestricted non-repudiation is an undesirable property for secure messaging protocols [BGB04]. Non-repudiation can be considered to enable a form of insider attack: a protocol participant can cryptographically prove to a third party that their communication partner sent a message, without the consent of that communication partner. This vulnerability is not present in unencrypted conversations; the attack is only possible due to the presence of non-repudiable cryptographic proofs (e.g., digital signatures). At first glance, it may seem that this is a necessary trade-off in order to provide authentication to the clients, but the situation is actually more nuanced: if a secure messaging protocol can somehow authenticate participants in a non-transferrable way (i.e., using a proof mechanism that only provides evidence to a designated verifier), then it is possible to achieve authentication without enabling this insider attack. Protocols with this ability to plausibly repudiate message authorship are said to be *deniable*. Such protocols are desirable because they allow participants to send messages "off-the-record", as they would with unencrypted protocols.

Deniability is a notoriously difficult concept to define. This problem arises due to the fact that deniability is actually a series of distinct, but related, properties. Deniability must be discussed with respect to an action and a type of judge.[4] An action is deniable with respect to a given judge if the judge cannot be convinced that an individual performed the action. To make such a definitive statement about a judge's behavior, one must define the type of evidence that is required to convince the judge that the action was performed, and the environment in which the judge resides. If an action is deniable with respect to a judge, then individuals can "plausibly deny" performing the action (with the definition of "plausibility" being determined by the requirements of the given judge).

It is not possible for a secure messaging protocol to afford more plausible deniability to message authors than an unencrypted protocol operating over the same network. This is because it is always possible for a participant to provide the plaintext transcript corresponding to a secure messaging session to a judge, regardless of the secure messaging protocol in use; this provides the judge with the same set of evidence that they would receive in the unencrypted case. Consequently, meaningful statements about deniability for secure messaging protocols must show that the protocol is not *worse* than the unencrypted case. To eliminate any possible gap, deniability proofs formulate the judge as an entity that would not be convinced by a plaintext

[4] Herein, "judge" refers to an entity that decides whether or not a certain event occurs. This entity is not necessarily a "judge" in the legal sense—in practice, a judge may be an ordinary citizen trying to ascertain the validity of a claimed message.

transcript alone: a perfectly rational decision maker that accepts only cryptographic arguments as evidence.

Deniability for secure messaging can be divided into two types based on the judge's environment: *offline deniability* and *online deniability*. Offline deniability is a more traditional property that can be found in protocols like the first version of OTR [BGB04], while online deniability has only just begun to appear in secure messaging protocols like the fourth version of OTR [OTR17] as a result of the work in Chapter 5. Both are forms of insider security.

An offline judge examines the transcript of a protocol execution that occurred in the past, and decides whether or not the event in question occurred. A judge of this type is given a purported *protocol transcript*, showing all of the (usually encrypted) data transmitted between participants, and a *chat transcript*, showing the high-level chat messages that were exchanged. The judge must then decide whether the protocol and chat transcript constitute proof that the action in question occurred (e.g., a given user sent a given message, or two given users communicated with each other using the secure messaging protocol). When proving the deniability of protocols, it is also normally assumed that an offline judge is given access to the long-term secrets of all parties named in the transcript; judges should not be able to distinguish real transcripts from fake ones even when given access to these secret keys. Since a judge with access to these long-term secrets has at least as much distinguishing power as the same judge without this access, designing protocols that achieve this level of deniability ensures that judges have no incentive to compromise long-term secrets in practice.

Typically, deniability with respect to offline judges is provided by designing the protocol such that many potential entities can produce forged chat and protocol transcripts. An offline deniability security proof defines a simulator program that produces transcripts, and then shows that no judge can distinguish these from real transcripts. When the simulator requires fewer inputs to produce a forgery, more real-world entities can run the simulator, and thus the deniability becomes more plausible.

An online judge interacts with a protocol participant, referred to as the *informant*, while the secure messaging protocol is being executed. The judge has a secure and private connection to the informant, and may instruct the informant to perform actions in the protocol. In practice, the judge may be coercing the informant into participating in this attack, such as a government agency forcing the covert cooperation of a service provider. The goal of the judge is to evaluate whether the actions of other participants in the protocol are actually occurring, or if the informant is fabricating the conversation (i.e., they are actually a *misinformant*). The judge does not have any direct visibility into the network, but it may instruct the informant to corrupt participants. The judge is also informed whenever a participant has been corrupted. This situation can be

likened to a real-world situation in which the informant is "wearing a wire" and an earpiece providing secure communication to a judge in another physical location.

A protocol offers *strong deniability* if it offers both offline and online deniability. If a secure messaging protocol claims to offer insider security, then it must also provide strong deniability or explicitly consider this type of insider attack to be out of scope.

2.5 DAKEs in Secure Messaging

Deniability for secure messaging protocols is primarily implemented as part of the initial Authenticated Key Exchange (AKE) sub-protocol. An AKE is a protocol that allows two parties—an initiator and a responder—to securely derive an ephemeral shared secret and authenticate each other using long-term *identity keys* without the assistance of a trusted authority. Bellare and Rogaway first formalized the definition of AKEs in 1993 [BR93a]. Shortly afterward, several AKEs claimed to offer deniability informally [Kra96; Kra03; BMP04]. Each of these Deniable Authenticated Key Exchanges (DAKEs) lacks some aspect of strong deniability. Contemporaneously, deniability was also widely studied in the context of authentication schemes [DNS98; DDN00; Kat03; YLP11].

With the release of the OTR protocol in 2004, deniability was recognized as a desirable feature for secure messaging [BGB04]. Since then, a variety of DAKEs have been published [DGK06; JS08; DKSW09; YZ13; WWX14; Sch15; UG15; MP16; UG18]. Walfish [Wal08] was the first to introduce a DAKE, Φ_{dre}, that simultaneously provides strong deniability, (weak) forward secrecy [BPR00], security against active attackers, and operation without trusted authorities. This work was later reiterated in a publication by Dodis et al. [DKSW09]. Unger and Goldberg [UG15] subsequently introduced two DAKEs designed for secure messaging—RSDAKE and Spawn—with comparable security proofs. Notably, Spawn was the first DAKE with (partial) online deniability that can be used in non-interactive applications. Unfortunately, RSDAKE and Spawn have several limitations that make them less desirable for practical deployments, including the use of obscure and inefficient cryptographic primitives.

OTR [AG07] and Signal [Ope13], the two most popular secure messaging protocols that use DAKEs, do not provide strong deniability. OTR's variant of the SIGMA DAKE offers no online deniability, and requires fragments of legitimate exchanges to forge transcripts offline. Signal originally used Triple Diffie-Hellman (3DH) [Mar13], an implicit DAKE with unrestricted offline deniability (i.e., anyone can forge transcripts using only public keys), but lacking online deniability. Signal subsequently switched to a DAKE known as Extended Triple Diffie-Hellman (X3DH) [MP16] that improves forward secrecy but regresses to the deniability properties of

OTR's DAKE. Consequently, real-world deployments of "deniable" secure messaging protocols still lack strong deniability. Chapter 5 presents two new DAKEs that provide strong deniability for secure messaging while overcoming the limitations of RSDAKE and Spawn.

2.6 Secure Messaging Design Layers

While most complete secure messaging solutions try to deal with all possible security concerns, secure messaging can be divided into three nearly orthogonal design layers: the *trust establishment* problem, ensuring the distribution of cryptographic long-term keys and proof of association with the owning entity; the *conversation security* problem, ensuring the protection of exchanged messages during conversations; and the *private transport* problem, hiding the communication metadata. The following subsections examine each of these design layers in turn.

While any concrete tool must decide on an approach for each design layer, abstractly defined protocols may only address some of them. Additionally, the distinction between these three design layers is sometimes blurred since techniques used by secure messaging systems may be part of their approach for multiple design layers. For each design layer, a given protocol can be analyzed in terms of its security and privacy properties, the usability impacts, and additional barriers to adoption.

The three design layers—trust establishment, conversation security, and private transport—and their associated properties are discussed in sections 2.6.1, 2.6.2, and 2.6.3, respectively.

2.6.1 Trust Establishment

One of the most challenging aspects of messaging security is *trust establishment*, the process of users verifying that they are actually communicating with the parties they intend. *Long-term key exchange* refers to the process where users send cryptographic key material to each other. *Long-term key authentication* (also called *key validation* and *key verification*) is the mechanism allowing users to ensure that cryptographic long-term keys are associated with the correct real-world entities. *Trust establishment* refers to the combination of long-term key exchange and long-term key authentication. After contact discovery (the process of locating contact details for communication partners using the messaging service), secure messaging tools must perform trust establishment to facilitate secure communication. Without secure trust establishment, it is always possible for an active adversary (such as the service provider) to violate the confidentiality of

conversations by sending the wrong long-term keys to the participants, establishing a conversation with each participant under a false identity, and relaying messages between them.

Most trust establishment schemes require key management: secure messaging tools must generate, exchange, and verify other participants' keys. For some approaches, users may be confronted with additional tasks, as well as possible warnings and errors, compared to classic tools without end-to-end security. Unfortunately, as discussed in Section 2.1.2, usability studies have consistently shown that most end users cannot reliably perform trust establishment that involves visible key management. Moreover, they lack appropriate mental models to understand the objective of trust establishment or why it is necessary, even for systems that enable them to complete the trust establishment ceremony correctly. This is unfortunate because, from a philosophical standpoint, the correct "identity" that a cryptographic key should be bound to is a construct within the user's mind, and so it is difficult for a secure messaging tool to perform long-term key authentication without user input. More concretely, in one scenario, Alice might expect "Bob's" private key to be accessible only to a human she has previously met, whereas in another scenario, Alice might expect "Carol's" private key to be accessible only to the operator of a particular social media account. Moreover, the notion of "Bob" and "Carol" may evolve in Alice's mind (e.g., if Alice meets Carol in person). It is supremely difficult to design usable trust establishment systems that can be completed reliably, much less systems that can evolve with users' notions of identity.

In practice, secure messaging tools that are designed to protect low-risk users from mass surveillance can focus on passive adversaries. For these systems, it makes sense to implement a simple long-term key exchange and automatically monitor for misbehavior such as key equivocation (i.e., if the service provides conflicting information about the long-term key for a given user) without having the user perform explicit long-term key authentication. High-risk users are targeted by active adversaries, so tools often implement additional, optional long-term key authentication methods to serve their needs.

2.6.2 Conversation Security

After *trust establishment* has been achieved, users of a secure messaging protocol can begin communicating. Each message that is sent using a secure messaging protocol can be thought of as belonging to a *conversation* associated with a list of participating *members* that are authorized to receive the message. Secure messaging protocols can structure these conversations in a wide variety of ways: protocols may impose limits on the number of members in the conversation (e.g., "direct messaging" protocols may limit membership to exactly two), associate conversations with chat histories, alter the membership of a conversation over time, structure the secure

messaging tool's user interface in terms of conversations, automatically create conversations when messages are sent, apply an ordering to messages in a conversation, and more. This notion of a "conversation" in terms of a set of messages, each associated with a set of authorized recipients, is handled quite differently by secure messaging tools with different communication models, such as two-party instant messaging, chat rooms, group teleconferences, email-like asynchronous message delivery services, and microblogging platforms. Regardless of how a secure messaging protocol creates and manages conversations, the protocol's *conversation security* scheme protects the security and privacy of the exchanged messages. This encompasses how messages are encrypted, the data and metadata that messages contain, and what cryptographic protocols (e.g., ephemeral key exchanges) are performed. A conversation security scheme does not specify a trust establishment scheme nor define how transmitted data reaches the recipient.

In classic messaging tools, users must only reason about two simple tasks: sending and receiving messages. However, in secure messaging, additional tasks might be added. Old systems, often based on OpenPGP, allow users to manually decide whether to encrypt and/or sign messages. Most recent tools secure all messages by default without user interaction. However, other usability and adoption factors, such as resilience to poor network conditions, should be taken into account. Modern approaches to conversation security achieve advanced security properties beyond the traditional confidentiality, integrity, and authentication guarantees: forward secrecy, post-compromise security, anonymity preservation, deniability, and asynchronicity are examples of some properties discussed in the literature. These properties were defined by Unger et al. [UDB+15] as follows:

- **Confidentiality**: Only the intended recipients are able to read a message. Specifically, the message must not be readable by a server that is not a member in the conversation.
- **Integrity**: No honest conversation member will accept a message that has been modified in transit.
- **Authentication**: Each member in the conversation receives proof of possession of a known long-term secret from all other members that they believe to be participating in the conversation. In addition, each member is able to verify that the message was sent from the claimed source.
- **Forward secrecy**: An active adversary that captures encrypted messages and subsequently corrupts all key material cannot decrypt the previously captured messages. The forward secrecy property for a particular protocol might be more limited about the adversarial capabilities. For example, protocols may specify that the forward secrecy property holds only when the delay between when the captured messages were sent and when the adversary corrupts the keys is longer than a given duration.

- **Post-compromise security**: A passive adversary[5] that corrupts all key material (thereby allowing it to decrypt newly sent messages) will eventually become unable to decrypt newly sent messages. Similarly to the forward secrecy property, the post-compromise security property usually only restores confidentiality for new messages after some given amount of time has elapsed after the adversarial key compromise.

- **Anonymity preservation**: Any anonymity features provided by the underlying private transport layer (discussed in Section 2.6.3) are not undermined. For example, if the private transport layer ensures that message senders are anonymous, then the conversation security layer does not deanonymize senders by attaching identities to message ciphertexts.

- **Deniability**: It is possible to plausibly deny participation in a conversation. Deniability can be divided into *offline deniability* and *online deniability* types. Definitions for these properties were given in Section 2.4.

- **Asynchronicity**: Messages can be sent securely to disconnected members and received by those members upon reconnection.

There are many security and privacy design issues specific to secure group chat protocols. For example, if groups have dynamic membership, then members must not be able to decrypt messages sent before they join or after they leave. There should be a mechanism to ensure that all members of the group perceive the same participant list (and possibly the same messages) to exclude the possibility of an invisible adversarially controlled member. The conversation security protocol should provide cryptographic enforcement of invitations, moderation, and member eviction, if applicable. For some applications, it is also desirable to guarantee that all participants receive messages in the same order (or at least in an order that preserves the causality of replies). There are many additional properties that are specific to group settings. Chapter 7 enumerates the conversation security goals for the new secure group messaging protocol introduced in this work. Conversation security for group settings can be broken down into even more granular design layers if desired [Wei19].

[5] No defenses in the conversation security layer can provide post-compromise security against an active adversary, because such an adversary can always use the compromised key material to behave in the same way as the compromised conversation members would. The only way to recover against such an attack would be to invoke the trust establishment scheme to authenticate some replacement keys. Secure messaging protocols in the literature normally discuss post-compromise security in terms of a passive adversary, while secure messaging tools usually provide a way for the users to perform trust establishment again if a compromise is suspected.

2.6.3 Private Transport

The private transport layer defines how messages are exchanged, with the goal of hiding message metadata such as the sender, receiver, and conversation to which the message belongs. The most generic type of private transport architecture simply adds privacy to data links between entities; this type of architecture can be combined with any conversation security architecture. However, the private transport architectures that offer the greatest privacy protections usually impose some sort of structure on the conversation security layer. For example, a private transport architecture might require that every user in the system operates a "mailbox" server that can only be reached through a combination of specific peer-to-peer network protocols, or it might require that all messages must be sent as part of globally visible batches that occur at regular intervals. When using these more restrictive private transport architectures, the techniques in the conversation security layer must take the required structure into account. The private transport schemes may also be used for privacy-preserving contact discovery.

Of the three design layers, private transport has remained the most difficult to solve. Recent work has shown that there are inherent insurmountable conflicts between strong anonymity, low latency overhead, and low bandwidth overhead, even in protocols where participants work together to enhance anonymity [DMMK20]. Nonetheless, anonymity remains an important property, especially for high-risk users—former National Security Agency director Michael Hayden famously stated that "we kill people based on metadata" [Joh14, 17:54]. Several secure messaging protocols have attempted to find an acceptable compromise between usability and anonymity by using sophisticated private transport mechanisms.

Private transport is not the focus of this work because it is deeply challenging and largely orthogonal to trust establishment and conversation security. However, it is not entirely orthogonal: it is possible for a protocol to undermine the guarantees of a private transport mechanism. For example, a protocol that includes user identifiers in plaintext as part of its conversation security layer will undermine network-level anonymity by revealing conversation metadata. A conversation security design that does *not* undermine the guarantees of private transports is said to have *anonymity preservation*. Thus, a conversation security design with anonymity preservation is necessary but not sufficient for anonymity: it must be combined with an appropriate private transport scheme if anonymity is desired.

2.6.4 Other Considerations

There are several other considerations that should inform the design of a complete secure messaging protocol. These aspects are mostly independent of the other components, and do not fit neatly into the previous design layers.

When secure messaging is used in an *instant messaging* setting (i.e., an environment in which messages are sent and received with low latency), it is important to provide *presence information*. Presence information includes data about a user's messaging state, such as whether they are currently "online" (available to receive interactive messages), a custom textual status message, user-provided information about their current activities (e.g., music or games being played), and more. Traditional insecure messaging applications typically provide presence information about a user's contacts. This information provides improved quality of service, but it is also important for improved protocols. Specifically, knowing whether a communication partner can conduct interactive protocols enables dynamic selection of the transmission mechanism.

Unfortunately, naïvely storing presence information on a central server exposes metadata, potentially undermining the properties of the private transport design layer. Preserving presence privacy requires more sophisticated techniques like those of the DP5 protocol [BDG15], which uses Private Information Retrieval (PIR) to distribute presence information without revealing contact relationships through access patterns. Cobb et al. [CSKH20] studied the features and implementation of presence information across 40 mobile applications and the implications for metadata privacy. They derived design guidelines and concepts for privacy-conscious systems that provide presence information.

Another important consideration is how to handle users that own multiple devices that must all be given access to the secure messaging system. Handling this problem (or ignoring it) has implications for all of the design layers, with particularly important implications for the trust establishment mechanism. The simplest approach, which is used by the majority of the popular secure messaging tools, is to generate a unique long-term key pair for every device, and to maintain a list of long-term public keys associated with each "account" (e.g., entity uniquely identified by a username). The downside of this approach is that the list of user devices is exposed to contacts, and trust establishment must be performed for each device. Moreover, changes to a user's devices (e.g., when a device is purchased, sold, or stolen) necessitates new trust establishment activities for every one of that user's contacts.

Shatter [AH16] investigated the use of threshold cryptography to solve the multi-device problem in the context of encryption and signature generation. This work also enumerated other possible approaches, including key synchronization between devices, the use of a "personal PKI" (a master certification keypair that authenticates device keys), and group signatures. Each of these approaches has advantages, disadvantages, and implications for each protocol design layer.

2.7 Two-Party Secure Messaging

Before considering secure group messaging, it is beneficial to trace the history of two-party protocols. While the two-party scenario is simpler, both in terms of the desired security properties and the cryptographic schemes, many of the techniques and lessons learned are applicable to the group setting. This section summarizes some of the most significant two-party secure messaging protocols.

The most basic features that a secure messaging protocol can provide are confidentiality and integrity. These properties can be achieved by simply using participants' static long-term asymmetric keypairs for signing and encrypting. OpenPGP and S/MIME are two well-known and widely implemented standards for message protection, mostly used for email but also in tools based on the Extensible Messaging and Presence Protocol (XMPP) [CDF+07; FL04; RT10; Sai11].[6] While most OpenPGP and S/MIME tools are vulnerable to surreptitious forwarding [Dav01] or identity misbinding [DOW92] attacks, these can be prevented in a new protocol instantiation by using an appropriate authenticated encryption scheme. The main weaknesses of this approach are that it lacks forward secrecy (i.e., compromise of a secret key enables retroactive decryption of recorded ciphertexts encrypted for the matching public key), it produces cryptographic proof of conversations (i.e., it is non-repudiable and lacks deniability), and it must be combined with more complex mechanisms to achieve the nuanced security properties introduced in later schemes.

Several authors have designed secure messaging systems using Identity-Based Encryption (IBE) in order to simplify key distribution. IBE allows a user's public key to be derived from their identifier and a master public key that is distributed as part of the setup. Unlike traditional systems that use public keys *as* identifiers (e.g., Ricochet [Ric14]), IBE-based schemes allow users to choose and distribute arbitrary identifiers without explicitly attached public keys. One of the earliest and simplest of these IBE-based secure messaging systems is SIM-IBC-KMS [BMHD08], which acts as an encryption overlay on top of the (now discontinued) MSN chat network. Messages are encrypted using IBE with a third-party server acting as the Private Key Generator (PKG). The protocol from Wang et al. [WLL13] operates similarly, but distributes the PKG function across multiple servers with a non-collusion assumption. The main problem with these approaches is that while they simplify key distribution (since public keys can be derived from identities), they do not simplify trust establishment. Specifically, users still need to somehow authenticate

[6] XMPP is a popular federated protocol for instant messaging that is often used in Internet telephony applications. The XMPP open standard also provides mechanisms for distributing presence information (e.g., whether users are currently online) and maintaining contact lists. Secure messaging protocols can be layered on top of XMPP to provide end-to-end encryption; the Gajim [FL04] messaging client, for example, can encrypt messages using several different protocols, including OpenPGP, before sending them with XMPP.

that the identities correspond to the expected real-world entities using one of the other trust establishment techniques. In practice, this means that IBE schemes do not have any significant advantage over traditional key distribution methods. Moreover, IBE-based schemes introduce a vulnerability: if the PKG is compromised (or some threshold is compromised in the distributed variant) then all authentication is undermined.

A more interesting potential use of IBE for secure messaging is *forward secure IBE* (FS-IBE). One example is the scheme proposed by Canetti, Halevi, and Katz [CHK03]. FS-IBE schemes permit efficient distribution of ephemeral public keys; the associated private keys can be erased to preserve forward secrecy. *Puncturable encryption* [GM15] is a similar mechanism that provides forward secrecy by updating a private key so that it can no longer decrypt ciphertexts with a specific tag.

The oldest and simplest way to achieve forward secrecy is to use a central key directory to distribute ephemeral public keys. This technique has efficient implementations based on well-tested security assumptions, but lacks the asymptotic efficiency (and thus scalability) of FS-IBE or puncturable encryption. Two early schemes in which users upload ephemeral keys to a key directory are IMKE [MO06] and SIMPP [YK07; YKAL08]. To prevent denial-of-service attacks, the key directory authenticates users before accepting new ephemeral keys. IMKE uses passwords for authentication, while SIMPP uses long-term keys and digital signatures. These protocols both lack end-to-end confidentiality because the key directory can inject malicious ephemeral keys.

While the use of central servers for presence information and central authentication is fundamental to systems such as IMKE and SIMPP, there is an alternative class of solutions that instead perform end-to-end authenticated Diffie-Hellman (DH) AKEs. The session key output by the AKE is used to derive symmetric encryption and Message Authentication Code (MAC) keys, which then protect messages using an encrypt-then-MAC approach. This basic design provides confidentiality, integrity, and authentication. This process does not inherently require a third party, but it still requires a trust establishment mechanism for the long-term keys. TLS with an ephemeral DH cipher suite and mutual authentication (TLS-EDH-MA) is a well-known example of this approach that uses the TLS Public Key Infrastructure (PKI) for trust establishment. Note that further protections, such as the party identifiers included in the SIGMA protocols [Kra03], are required during the key exchange to protect against identity misbinding attacks.

OTR [BGB04] uses ephemeral session keys derived from an AKE for conversations, and introduces a number of additional mechanisms designed to support deniability. The use of ephemeral session keys provides forward secrecy and post-compromise security (i.e., compromising session keys does not compromise future sessions) between conversations. This technique also makes message authorship unlinkable (i.e., since different conversations do not use the same keys, one cannot use knowledge of authorship in one conversation to derive knowledge of authorship for

another session). Message authorship is deniable against offline judges since a DAKE is used, and messages are authenticated with shared MAC keys rather than being signed with long-term keys. The simulator corresponding to the DAKE used by OTR requires a legitimately signed ephemeral public key as input. This means that messages can be forged by any entity that can establish real connections to the claimed participants to receive the required signatures. OTR users can increase the number of possible forgers by publishing previously signed ephemeral keys in a public location, thereby providing the appropriate simulator input to forgers. OTR additionally publishes old MAC keys and intentionally uses malleable encryption, which expands the set of possible message forgers by enabling protocol transcripts to be altered undetectably [BGB04].

A desirable property is forward secrecy for individual messages rather than for entire conversations. This is especially useful in settings where conversations can last for the lifetime of a device. To achieve this, the session key from the initial key agreement can be evolved over time through the use of a *session key ratchet* [PM16]. A simple approach is to use a Key Derivation Function (KDF) to compute future message keys from past keys. This simple approach, as used in SCIMP [MBZ12], provides forward secrecy. However, it does not provide post-compromise security within conversations: if a key is compromised, all future keys can be derived using the KDF.

A different ratcheting approach, introduced by OTR, is to attach new DH contributions to messages [BGB04]. With each sent message, the sender advertises a new DH public key. Message keys are then computed from the latest acknowledged DH values. This design introduces post-compromise security within conversations since a compromised key will regularly be replaced with new key material. A disadvantage of the DH ratchet is that session keys might not be renewed for every message (i.e., forward secrecy is only partially provided).

The Signal protocol [Ope13] adopts many of OTR's techniques and builds upon them for use in mobile environments, where devices are frequently offline and conversations last for a very long time. Signal uses a *double ratchet* [PM16; UDB+15; CCD+17; BRV20] for key evolution. This technique combines the DH and KDF ratchets of OTR and SCIMP. DH contributions attached to messages are used to instantiate new SCIMP-style ratchets. Each stage of these KDF ratchets is hashed to derive the key for the individual message. This technique permits messages to be dropped or arrive out of order while still preserving per-message forward secrecy and providing post-compromise security across DH ratchet stages. Signal also incorporates an implicit AKE called X3DH [MP16]. This AKE is performed between long-term keys, a digitally signed medium-term (e.g., weekly) ephemeral key, and one-time ephemeral keys. The ephemeral keys for a message receiver, called the *signed prekey* and *one-time prekey*, are distributed to senders by an untrusted central server. This construction provides better forward secrecy properties, prevents the server from undermining confidentiality, and enables asynchronous messaging (the ability to send messages to offline users).

2.8 Secure Group Messaging

Secure group messaging is a more difficult problem than two-party secure messaging. Compared to two-party secure messaging, there are fewer academic papers and far fewer end-user tools. This section presents a comprehensive survey of notable systems and their distinctive advantages.

2.8.1 Centralized Schemes

One of the earliest secure group messaging schemes available to end users was SILC [SILC00]. The SILC project has published a wire protocol and an official tool, and the protocol has been briefly analyzed in an academic context [SYG08]. SILC most closely resembles the IRC protocol with an encryption layer. Messages are routed to channels across federated servers using client-to-server encryption for metadata (but without network path obfuscation). Messages to channels are also encrypted using AES in an encrypt-then-MAC mode. By default, the key for message encryption is known to the servers. However, there is an optional mode in which users can provide their own pre-shared key when joining a channel. This very basic design allows SILC to scale well, but the usability of opt-in pre-shared keys is poor, and the central servers retain the ability to launch myriad attacks (e.g., message replays, manipulation of channel lists, and traffic analysis).

In an early and widely cited result, Kikuchi et al. [KTN04] designed a protocol that resembles a group version of SIMPP [YK07], although it was published several years before SIMPP. Users register long-term keys with a fully trusted central server that forms channels and acts as a key directory. Users establish two-party session keys using an unusual DH-based key exchange protocol that provides authentication if the server is not malicious. The creator of a channel generates and distributes a room key using two-party session keys. Everyone encrypts their messages using this room key. In addition to breaking confidentiality by exploiting the key directory, a malicious server can perform the same attacks as a SILC server.

The GROK [CKFP10] system is a plugin for the Pidgin instant messaging client that enables secure group messaging. Like the aforementioned schemes, GROK uses pairwise AKEs to distribute a room key. Long-term public keys are exchanged out-of-band. Unlike the previous schemes, GROK refreshes the room key when the group membership changes; this prevents insiders from decrypting captured messages sent when they are not part of the group. GROK accomplishes this by selecting a user to distribute a new room key using the minimum number of pairwise channels.

The "group OTR" protocol from 2007 [BST07] is another extremely simple scheme. One group participant acts as a proxy that controls all of the messages. Other members initiate

OTR conversations with the leader and use these channels to send messages to the group. Like SIM-IBC-KMS, this scheme was designed to use the MSN network for message transport.

Thukral and Zou [TZ05] published the first academic group messaging paper to use identity-based cryptography, but the design suffers from numerous cryptographic deficiencies. Their IBECRT system assumes a fully trusted central server that routes messages, maintains presence information, and acts as the PKG. Users register with the server using a username and password. A user's identity is an MD5 hash of their username. When logging in, a user encrypts and sends their password to the server, which then replies with their private key encrypted using a key derived from their identity and password. Group conversations are formed by assigning co-prime scalars to each user. The group initiator selects the initial list of group members and broadcasts ciphertexts formed using each group member's scalar. These ciphertexts can be decrypted using the Chinese remainder theorem and the receiver's private key, allowing each member to verify the list of participants (and thus the protocol offers *participant consistency*). Finally, each participant sends a key contribution to the others such that everyone can derive the same session key for the conversation. This initial key establishment requires $O(n^2)$ computation and transmission for n users. When a new user joins the group, their contribution can be directly incorporated into the session key, permitting joins with $O(n)$ work. When a user leaves the group, the protocol must essentially restart.

2.8.2 Causality Preserving Schemes

Unlike the aforementioned work, which is typically concerned with group key establishment and membership management, the OldBlue [VC12] protocol instead focuses on reliability for the subsequent messages. Specifically, OldBlue is concerned with *causality preservation*: enabling tools to avoid displaying a message before other messages that causally precede it. OldBlue also provides *speaker consistency*: all participants agree on the sequence of messages sent by each participant. Each message in OldBlue is encrypted under a previously established group key and signed with the sender's long-term key. Each message includes a minimal list of message identifiers that causally precede it (i.e., messages that were observed by the sender before they sent the message). An OldBlue message identifier is a hash of the message contents, the sender, and the list of preceding identifiers. In this respect, OldBlue message identifiers form a hash tree. When a client receives a message with a missing predecessor, it internally buffers the message and issues resend requests to all other group members. Luo [Luo14] examined this UI policy, proposed and evaluated several different ways of handling out-of-order messages, and described mutual exclusion between various forms of transcript consistency.

KleeQ [RKAG07], a protocol designed for use by multiple trusted participants with tenuous connectivity, uses a different approach for causality preservation. The protocol begins with a deniable authenticated multi-party key exchange,[7] which establishes a shared secret key among the participants. The group can be efficiently expanded by incorporating the DH contribution of a new member and deriving a new group key. However, like IBECRT, the protocol must be restarted when a participant leaves in order to preserve confidentiality. For deniability purposes, all messages are encrypted and authenticated with a MAC using keys derived from the group secret. When two participants can establish a connection, they exchange the messages that the other is missing using a patching algorithm. The participants deterministically partition the sequence of messages into a sequence of "blocks", each containing multiple messages. The KleeQ protocol guarantees that any message in a block is not a reply to a message in a future block—it may be a reply to messages in the same or previous blocks, or it may not be a reply at all. The participants "seal" a block by verifying that they all agree upon its hash. After each block is sealed, the participants derive new keys using the previous keys and the block contents. KleeQ includes a mechanism to seal blocks even if some users are inactive in the conversation. This occasional block sealing mechanism ensures that causality is preserved and that participants have a consistent view of the conversation (e.g., it prevents users from sending different messages to other participants). This mechanism achieves the same purpose as the message predecessor lists in OldBlue.

2.8.3 Comprehensive Decentralized Schemes

The mpOTR [GUVC09] scheme is a highly influential approach for secure group messaging, although it was never defined as a wire protocol or implemented. mpOTR is designed for use by a group of interactive participants with a fixed and known group membership list. Like OTR, mpOTR includes many mechanisms meant to preserve various notions of deniability. In the original scheme [GUVC09], participants initially perform pairwise DAKEs and use these deniable channels to distribute ephemeral verification keys for a digital signature scheme. In follow-up work [Van13b], this procedure was replaced with a more efficient protocol that derives a shared room key for all participants and distributes ephemeral verification keys. During the conversation, messages are encrypted with the room key and signed with the sender's ephemeral signing key. Once the conversation has finished, a special "shutdown phase" ensues. Each participant broadcasts a hash of all messages seen in the conversation, in a well-defined order. If there is a hash mismatch, the user is alerted that an inconsistency has been found in the conversation, but no specific information about the anomaly is available. A significant drawback of this approach is that in many deployment scenarios, there is no notion of "finishing" a conversation.

[7] ^ Group key exchanges are discussed in detail in Chapter 3.

Signal [Ope13] offers group messaging support in addition to its prolific two-party protocol. While the project originally considered adopting mpOTR, this idea was rejected for several reasons [Mar14b]. Specifically, mpOTR is not well suited for mobile environments due to its setup overhead, short-lived conversations, lack of in-conversation forward secrecy, delayed and coarse transcript consistency notifications, and implementation complexity. Consequently, group messaging in Signal simply involves pairwise two-party Signal messages. When a message is sent to the group, the sender uses hybrid encryption with a KEM transmitted to each recipient using the two-party Signal protocol. The server duplicates the DEM for all recipients. The hybrid encryption key is reused across several messages for efficiency. Finally, the protocol also incorporates an OldBlue-style transcript consistency mechanism. This construction inherits many of the benefits of the underlying two-party Signal protocol, but lacks more advanced features like deniability, protection against equivocation, and participant consistency. The requirement to establish pairwise shared keys between participants using the KEM also means that the protocol scales poorly and is therefore not suitable for large groups. Moreover, the Signal protocol is no longer openly documented.

Liu et al. [LVH13] defined a protocol, BD-GOTR,[8] with the goal of improving the deniability properties of group messaging beyond those offered by mpOTR. Specifically, BD-GOTR provides strong deniability as defined in Section 2.4. Unfortunately, the protocol does not scale well due to its method of establishing shared group keys and the need to establish pairwise authenticated channels for transcript consistency verification. Chapters 3 and 4 describe more efficient key exchange techniques.

In subsequent work, Schliep et al. [SVH18] designed a new protocol, SYM-GOTR, with the same objectives as BD-GOTR. Unlike BD-GOTR's complex mechanism for establishing shared secret keys, SYM-GOTR simply establishes peer-to-peer deniable secure channels and distributes ephemeral keys over these channels. Each sender simply encrypts their message with a symmetric key that they have previously distributed using the pairwise channels. After a message has been broadcast, all users send a signed hash of the message over the pairwise deniable channels. This design imposes a consistent global transcript for all participants. The advantage of this design is that it provides very powerful security guarantees for small and low-traffic group instant messaging applications. The downside is that it scales poorly due to pairwise communication, and the entire protocol must halt during each message broadcast as the consistency checks are performed. The protocol cannot handle participants going offline, because all parties are needed for these consistency checks.

[8] ^ In the original publication, the protocol was named Group OTR (GOTR). However, this name was already used by an unrelated work in 2007 [BST07]. In a later publication [SVH18], the authors started referring to their earlier protocol as "BD-GOTR" for disambiguation.

2.8.4 Anonymous Conversations

There are a variety of abstract protocols and tools for secure group messaging that are primarily concerned with transport privacy. Anonymous group messaging applications require low bandwidth, low latency, and one or more forms of anonymity, depending on the intended use cases. Within these constraints, protocols offer several different security properties [UDB+15]. Senders, receivers, or both might be anonymous (from the perspective of a network adversary). Senders and/or receivers might be anonymous to other participants, with the anonymity set being equivalent to the group participant list. The protocols may or may not resist attacks by global adversaries, and they may or may not prevent or detect denial of service attacks or other misbehavior. Messages sent within one "conversation" might be unlinkable from the perspective of the network layer. Transport privacy schemes may also impose restrictions on the security properties (e.g., deniability) of the overall scheme. This section only discusses notable group messaging schemes that provide anonymity properties as their top priority; it does not include anonymity networks that are primarily intended for other purposes (e.g., web browsing).

One class of anonymous communication technique that is intrinsically well suited to group communication is DC-nets. DC-nets are interactive group protocols that operate in rounds. During each round, each member of the group either submits a secret message or no message. At the end of the round, every user receives the XOR of all submitted secret messages, but not the identity of the sender. When combined with encryption and a group key exchange protocol, these messages are only revealed to group participants. DC-nets require a mechanism to avoid message collisions. Anonycaster [Hea12] uses "silent rounds" to detect misbehavior. Dissent [CF10] and Verdict [CWF13] take a different approach by constructing a DC-net system through the use of a verifiable shuffle and bulk transfer protocol, which facilitates a blame protocol that can pinpoint the entity that caused a round to fail. Dissent appoints one participant as a leader to manage these systems. Verdict achieves better scalability by providing central servers that execute the DC-net protocol. As long as any one server is honest, users' privacy is maintained. Riposte [CBM15] and Atom [KCDF17] are more recent systems that scale to a large number of users, but require many minutes or hours of latency for message transmission; they are primarily intended for microblogging applications.

Mix networks [Cha81] are another popular anonymous communication technique that can be used for group communication while defending against global adversaries. Similarly to DC-nets, mix networks aggregate incoming messages into periodically published batches. Unfortunately, they also typically impose high latency or low throughput in order to achieve a desirable privacy level. While mix networks have historically been used primarily in anonymous email systems [SP06], they have recently found use in several private transport systems intended for incorporation into secure messaging protocols. Vuvuzela [HLZZ15] uses a mix network that

abandons cryptographic indistinguishability in favor of differential privacy [DMNS06]. This design achieves better performance than solutions based on DC-nets while still offering some protection against global passive adversaries. Stadium [TGL+17] subsequently improved upon this design by replacing components of Vuvuzela that required vertical scaling (i.e., more powerful servers to support more users) with components that scale horizontally (i.e., use more servers to support more users). In both cases, these schemes still require minutes of latency to transmit messages when scaling to millions of users in the system, making them more suitable for email or microblogging applications than for instant messaging.

Several other systems implement private transport by building on the Tor network [DMS04]. In Ricochet [Ric14], each user is associated with a Tor hidden service. Users periodically poll their server to see if any messages have arrived. Cwtch [Lew18] builds a group messaging layer on top of Ricochet channels. Creating a group in Cwtch produces a shared secret, group identifier, and invitations to securely distribute to other participants. Cwtch clients then anonymously subscribe to receive messages from the server, decrypt messages for their group(s), and perform an identity verification protocol over Ricochet channels. MTor [LSL16] instead extends Tor to support multicast communication by arranging relays in a tree structure, where each node forwards messages to its children. In MTor, a group chat is established by distributing a group descriptor that contains a group encryption key and a group signing key. MTor conversations take place over several sessions, where each session derives a different multicast structure and keys from the group descriptor; this prevents messages from being linked. Approaches based on the Tor network provide low latency, but make the schemes vulnerable to deanonymization attacks by global adversaries.

2.8.5 Contemporary Designs

The currently available tools for group messaging are all based on pairwise two-party protocols. Multiple public development efforts have attempted to produce a superior group messaging specification that scales to larger group sizes with stronger security properties.

The flute [Kad16] system is a very basic "full stack" group messaging scheme (i.e., it provides an abstract protocol, a wire protocol, and a tool) that resembles a simplified GROK. Although flute can operate on any chat substrate, the initial release is designed to be used over IRC. All users exchange long-term signing keys out of band. The room creator invites new users to the group. When a new user joins, they send a signed ephemeral key to the creator, who then generates a new group key, and sends it to all current members using their ephemeral keys. The protocol provides entity authentication, confidentiality, and forward secrecy, but it does not attempt to

provide deniability, transcript consistency, or participant consistency. Both the room creator and the underlying chat network have significant potential for malicious behavior.

The (n+1)sec [eQu15] protocol provides more advanced security properties than flute, but currently only exists as an abstract and wire protocol. The system consists of three sub-protocols: a deniable authenticated group key exchange, a communication protocol, and a transcript consistency verification protocol. The group key exchange results in a shared group key and the distribution of ephemeral signing keys for the participants. The protocol enables the computation of subgroup keys without further communication. The exchange concludes with a key confirmation step. Messages sent to the group are encrypted using the group's shared secret key and signed using the ephemeral signing key of the sender. Each message includes a new ephemeral key that causes the group key to ratchet forward. The protocol assumes that it operates on a reliable network that guarantees a global message ordering. A transcript verification, similar to the one in mpOTR, is performed after each message is sent to ensure that the global transcript is consistent. Joining or leaving the group is made efficient by the key agreement scheme, but requires synchronous communication. Optionally, members can informally forward messages to users as the joining procedure is being performed.

Unlike (n+1)sec, the mpEnc [LK16] protocol does not require the network substrate to impose a global order on messages, although it still requires group membership operations to be handled sequentially. Each group membership change begins a new "subsession" with a group key exchange. Though this exchange accomplishes the same goals as the one in (n+1)sec, it is non-repudiable and inefficient. Within a session, mpEnc implements OldBlue-like transcript consistency checks. The protocol also includes message acknowledgements that raise warnings when they time out. The authors provide an abstract protocol, wire protocol, and user interface design advice for handling transcript consistency. The mpEnc protocol was designed to be deployed as part of the chat feature on the MEGA file sharing website.

Schliep and Hopper [SH19] designed a deniable secure group messaging protocol named Mobile CoWPI that improves upon their previous work with BD-GOTR and SYM-GOTR, as discussed in Section 2.8.3. Mobile CoWPI is primarily concerned with providing deniability and transcript consistency in asynchronous environments: unlike BD-GOTR and SYM-GOTR, participants do not need to stay online during the conversation. Mobile CoWPI establishes pairwise secure channels using the NAXOS key exchange scheme [LLM07], which is nearly identical to 3DH and provides similar deniability properties. Conversation messages are encrypted using symmetric keys that are delivered in protocol messages. Protocol messages encrypt the new symmetric keys using the pairwise NAXOS shared secrets, as well as new ephemeral DH public keys to be used as part of new NAXOS exchanges—this constitutes a pairwise DH ratcheting mechanism. The system also incorporates "order-enforcing services" (OESes) that authenticate the position of protocol messages within the global transcript. As long as at least two OESes

are honest, Mobile CoWPI provides global transcript consistency. The main weakness of Mobile CoWPI is that sending a protocol message (e.g., an actual conversation message or a group membership alteration) involves transmitting a new key to each group member. This expense scales linearly with the group size, making the protocol unsuitable for larger groups.

In 2017, the disparate public efforts to design a secure group messaging protocol coalesced into the MLS project [BBM+20]. The goal of the MLS IETF working group is to specify, standardize, and implement widely deployable secure group protocol with all desirable security properties, including support for asynchronous communication without a trusted central server or designated "leader". MLS is by far the most comprehensive design to date, and understanding its design requires additional context discussed in Chapter 3. Chapter 4 describes the core primitive and design of MLS.

2.9 Chapter Summary

This chapter established the background knowledge necessary to understand the design of modern secure messaging protocols. Section 2.1 surveyed notable usability studies related to secure email and secure messaging; these studies are extremely helpful for determining what security properties to target when designing protocols. Section 2.2 defined different levels of abstraction for specifying secure messaging protocols, and the most appropriate use of each type. Section 2.3 provided a general definition of the threat model considered by the protocols in this dissertation. Section 2.4 presented common definitions of "deniability" with respect to secure messaging, including the distinction between "offline" and "online" deniability, with "strong deniability" encompassing both notions. Section 2.5 surveyed the most notable DAKEs used in secure messaging protocols; this background is very important for Part II, which presents new strongly deniable DAKEs for modern secure messaging applications. Section 2.6 defined several mostly orthogonal design layers that divide the task of developing a secure messaging tool into independently manageable problems. Section 2.7 surveyed notable secure messaging protocols for the two-party setting, and Section 2.8 did the same for the group setting.

This chapter introduced specification and design layers that can be used to frame a discussion about the design of secure messaging protocols, but it limited analysis of the surveyed protocols to a surface level. Chapter 3 introduces the specific background information necessary to analyze the conversation security layer of secure group messaging protocols, which is the focus of the primary contributions in Part III.

CHAPTER 3 | Group Key Exchanges

In this chapter: Background

In contrast to simple schemes built using pairwise secure channels, efficient secure group messaging systems typically involve the establishment of some shared secret key that is known to all participants. Aside from pairwise AKEs, the simplest cryptographic protocol that can be used to establish this "group key" is a Group Key Exchange (GKE). GKEs are also sometimes called group key agreement (GKA) protocols, along with a variety of other terms (e.g., conference key distribution systems). Moreover, some authors distinguish between the terms, defining GKAs to be contributory (i.e., every participant contributes to the group key) and GKEs to be non-contributory. For simplicity, all such schemes are hereafter referred to as GKEs with properties mentioned as needed.

In its most basic incarnation, a GKE is an interactive protocol performed between n parties (where $n \geq 2$, and the interesting cases occur when $n \geq 3$) that outputs a shared secret key. GKEs are usually designed to be secure (at a minimum, ensuring key secrecy) against a passive network adversary. The Katz-Yung compiler [KY03] trivially adds protection against active network adversaries if desired; this compiler adds digital signatures and nonces to an unauthenticated GKE to produce an authenticated GKE. The modifications made by the Katz-Yung compiler are typically suboptimal, so protocols can often be made more efficient by specifying a hand-crafted authentication mechanism.

GKEs assume that group membership is fixed when the protocol is performed. This limitation is inadequate for many higher-level protocols, so the GKE definition can be extended to support dynamic group membership. This is typically modeled by having an initial fixed-group "setup" phase, followed by "join" and "leave" sub-protocols to add and remove participants, respectively. Schemes supporting dynamic groups are widely referred to as Dynamic Group Key Exchanges (DGKEs). Note that these should not be confused with "deniable GKEs" (a GKE providing deniable authentication), for which no standard term exists. Some DGKEs additionally support optimized functions to add or remove multiple participants at a time, called "mass join" and "mass leave". Critically, any DGKE must update the group key after each membership change; this prevents new members from retroactively decrypting old messages or previous members reading messages

sent after their departure. Note that DGKEs are different from GKEs in a fundamental way: whereas GKEs are run once to derive a shared key and are then finished forever, DGKEs are stateful protocols that are executed indefinitely across multiple changes in the set of participants. Defining a DGKE therefore requires additional considerations about how state is stored and how this state interacts with "join" and "leave" operations over time.

DGKEs sufficiently met the demands of secure messaging protocol designers until recently, when the desire for asynchronicity, forward secrecy, and post-compromise security once again necessitated extending the definition. This chapter surveys notable GKE and DGKE schemes that can be used to build secure group messaging protocols, while Chapter 4 discusses subsequent extensions to the DGKE concept and their use in contemporary secure group messaging protocols like MLS.

3.1 Unobtainium: Non-Interactive Key Exchanges

An optimally efficient and maximally useful GKE would require only a single flow of communication and no a priori knowledge of key material. In such a scheme, each of the n participants would broadcast their independently generated key material, and upon receiving the other participants' material, everyone would immediately derive the same shared secret key. A scheme like this is incredibly useful: if key material is uploaded to a public bulletin board, then any participant would be able to derive a shared secret key for any subgroup at any time using a local computation with their private key as input. A GKE protocol that satisfies this special case is called a multi-party Non-Interactive Key Exchange (NIKE). The DH key exchange protocol can be viewed as a NIKE for the case where $n = 2$. Ideally, there would exist an efficient and secure generalization of DH to larger group sizes.

The only (arguably) practical NIKE realization for $n > 2$ is the Joux protocol [Jou00], which uses cryptographic pairings to solve the $n = 3$ case. The Joux technique, based on bilinear maps, can be naturally extended to the $n > 3$ case using multilinear maps [BS03]. The $n > 3$ case is also solvable using indistinguishability obfuscation [BZ14] or lattice assumptions [MZ17]. Unfortunately, the known approaches for $n > 3$ all rely on cryptographic techniques that are currently in a tumultuous "break and fix" research phase, or impose impractical resource requirements. Until these techniques become established and efficient, NIKEs are of limited use for secure group messaging, and we must instead rely on less desirable GKE constructions.

3.2 Early and Unique Schemes

In one of the first attempts to extend two-party DH exchanges to the group setting, Ingemarsson et al. [ITW82] presented multiple protocols. In one of their schemes, the n participants logically order themselves in a ring (i.e., each participant has two "neighbors"). The group key is derived in n rounds. Each participant begins with the group generator g as an intermediate value. During each round, each participant i raises their intermediate group element to their secret exponent x_i, then sends the element to their "next" neighbor. After the final round, the participants share the group key $g^{x_1 x_2 \cdots x_n}$. The main weakness of this approach is that it requires a large number of rounds.

The STR protocol [SSDW90] computes group keys based on DH exchanges between subgroups. Given a group X of participants with group secret key x and another group Y with secret key y, the group secret key for $X \cup Y$ is g^{xy}. Groups can be "merged" in this way by performing a DH exchange to compute the new shared secret. A group of n participants can derive a group key by performing a DH exchange between two participants, broadcasting a public key corresponding to their shared secret, performing another DH exchange between this first subgroup and a third participant, and repeating the process until all participants have been "merged". Note that this process requires an efficient mapping from group elements to scalars in order to derive the public keys and complete the exchanges. Moreover, this mapping cannot be the discrete logarithm, because if discrete logarithms were efficiently computable, the DH exchanges would not be secure. A hash of the binary representation of the group element is a suitable mapping. This STR approach would later be recognized as a special case of generalized DH tree-based approaches, where the STR exchange tree is a maximally unbalanced full binary tree of $n - 1$ exchanges [KPT02]. Section 3.5 discusses tree-based GKEs.

Becker and Wille [BW98] introduced several interesting schemes with the intention of improving efficiency. In the "Octopus" protocol, four of the n participants are selected as "leaders". These leaders derive a group key using three DH exchanges in a manner similar to a 4-party STR exchange. The remaining $n - 4$ participants each choose a leader and establish a shared secret with them. The leaders then distribute values to their followers that allow them to derive the group key. In the "2^d cube" protocol, each of the $n = 2^d$ participants becomes a vertex in a d-dimensional hypercube. During the i^{th} round, each pair of subgroups joined by an edge in the i^{th} dimension perform a 2-party DH exchange to form a new subgroup with a new subgroup key. After d rounds, all participants have been merged into a group with the same group key. The "2^d Octopus" protocol combines these approaches by replacing the four leaders in the Octopus protocol with 2^d leaders. While these schemes are interesting, the Octopus protocols require additional trust for the leaders, and all of the schemes are less efficient than subsequent designs.

Boyd and Nieto [BN03] described a very simple GKE that takes advantage of designating one trusted participant as the "leader". The $n-1$ "followers" each send their public key and a nonce to the group leader. The leader then chooses their own nonce, encrypts it for each public key, signs all of the communication data, and distributes these values along with all nonces to all followers. The group key is derived from the concatenation of nonces. The primary disadvantage of this scheme is that it grants complete control to the leader, making many insider attacks possible (e.g., the leader can cause followers to output different keys, or even choose arbitrary nonces to force a specific key selection). Essentially, this scheme is equivalent to a key distribution by a trusted authority.

Bresson and Catalano [BC04] described an unusual GKE based on a combination of ElGamal encryption and Shamir's secret sharing scheme. The core of the protocol involves each participant choosing a secret group element and encrypting it for each other participant. The group key is derived from the combination of all secret group elements; each participant can derive this key by combining ciphertexts (due to ElGamal being partially homomorphic) and decrypting the result. The secret sharing component is used as a sort of information-theoretically secure commitment scheme to prevent adaptive selection of group elements. The resulting GKE is provably secure in the standard model with the Decisional Diffie-Hellman Problem (DDH) assumption.

Yoneyama et al. [YYK+16] presented a DGKE based on techniques from the broadcast encryption and multicast literature. The protocol involves an honest-but-curious central server that assists in the key exchange. The primary concession of the protocol is that if the server is covertly malicious, it can completely compromise the DGKE. In exchange for this weakness, the DGKE supports many desirable security properties. The group key cannot be compromised using only long-term secrets or only ephemeral secrets. The scheme also requires only one round: messages sent from participants to the server, and a resulting broadcast from the server. The group key is efficiently changed regularly, and also when a member joins or leaves the group. Moreover, Yoneyama et al. also enable post-compromise security by periodically rotating the group key even when no group membership change has occurred. The scheme is constructed using ciphertext-policy attribute-based encryption, along with standard primitives.

3.3 The CLIQUES Family

The early proposed schemes from Ingemarsson et al. described in Section 3.2 operate by logically organizing participants in a ring, and all n participants send a constant-size message during each of the n rounds. Another early scheme based on a significantly different approach is the CLIQUES protocol suite [STW96; STW98]. The most notable protocol from the suite is called

GDH.2. In GDH.2, the participants are arranged in a line. The participant at the "end" of the line is called the "group controller" and should ideally be the most trusted participant. The participant at the "start" of the line initially "receives" the group generator g. When a participant receives a message, they exponentiate each element in the message with their secret key. The participant then sends the unmodified last element from the message they received, as well as all exponentiated values, to the next participant in the line. For example, the first participant will send $\{g, g^{x_1}\}$, the second will send $\{g^{x_1}, g^{x_2}, g^{x_1 x_2}\}$, the third will send $\{g^{x_1 x_2}, g^{x_1 x_3}, g^{x_2 x_3}, g^{x_1 x_2 x_3}\}$, and so forth. Once the group controller (the final participant) processes their incoming message, they will have a set of elements $\{g^{x_1 x_2 \cdots x_{n-1}}, g^{x_2 x_3, \cdots x_n}, g^{x_1 x_3 \cdots x_n}, \ldots, g^{x_1 x_2 x_3 \cdots x_n}\}$. This concludes the "upflow" phase of the protocol. In the "downflow" phase, the group controller sends to each participant the element missing that participant's secret in the exponent. All participants can then derive the group key $g^{x_1 x_2 \cdots x_n}$. This scheme may be useful in very niche networking scenarios where the unidirectional "upflow" phase is inexpensive, but it is very inefficient when performed over the Internet due to requiring n rounds of communication (albeit with a single participant in each round).

The CLIQUES suite is notable in that it was one of the earliest works to consider dynamic groups. When a new participant would like to join an established group, they become the new group controller. The old group controller generates a new secret key, and the protocol resumes the upflow phase from that point. If it is desirable for the role of the group controller to be fixed, then the protocol can be easily modified to insert new participants immediately before the existing group controller (rather than after it). When a participant is leaving an established group, the group controller generates a new secret key and performs a new downflow phase.

The original CLIQUES suite was unauthenticated (and thus insecure against active attackers) and its security was unproven. Bresson et al. [BCP01] modified GDH.2 to add authentication using digital signatures under long-term keys; they optimistically called the resulting scheme AKE1. They also proved the key secrecy and mutual authentication of the resulting scheme under the "group Computational Diffie-Hellman Problem (CDH)" assumption (G-CDH$_\Gamma$), which is essentially just the definition of the scheme itself. Specifically, the G-CDH$_\Gamma$ problem is to compute $g^{x_1 x_2 \cdots x_n}$ given just $\bigcup_{J \in \Gamma} g^{\prod_{j \in J} x_j}$. For the AKE1 security proof, Γ reflects the set of elements that are transmitted during the protocol. The same authors later defined a variant called AKE1+ [BCP02] that uses secure coprocessors for storing the secret keys and smart cards for producing digital signatures. They proved the security of AKE1+ in a stronger model (with notions of forward secrecy, key freshness, and AKE security) without random oracles, but with even more hardness assumptions.

3.4 The BD Family

One of the most prominent early GKEs that spawned significant follow-up research is the Burmester-Desmedt (BD) protocol [BD94]. Like the suite from Ingemarsson et al. described in Section 3.2, participants in the BD protocol are logically arranged in a ring. However, the BD protocol completes in only two rounds and with broadcast messages of constant size, making it much more suitable for use over the Internet than the n rounds of $O(n)$ sized messages used by other schemes. In the first round, each participant i generates a secret exponent x_i and broadcasts g^{x_i} to the group. In the second round, each participant computes $X_i = (g^{x_{i+1}}/g^{x_{i-1}})^{x_i}$, where the index wraps around the sequence $(1, 2, \ldots, n)$, and broadcasts the result to the group. The group key is $g^{x_1x_2+x_2x_3+\cdots+x_{n-1}x_n+x_nx_1}$. Each participant can compute the group key as $(g^{x_{i-1}})^{nx_i} X_i^{n-1} X_{i+1}^{n-2} \ldots X_{i-2}$. The original BD paper also includes an authenticated version of the scheme, which includes in the first broadcast a zero-knowledge proof of knowledge of discrete logarithms of both g^{x_i} and a long-term public key.

Kim et al. [KLL04], hereafter denoted Kim-Lee-Lee (KLL), simplified the BD scheme and extended it to dynamic groups, transforming it into a DGKE. The original BD scheme is used for the initial group setup with some modifications. Firstly, the second round broadcast is simplified: each participant now broadcasts $g^{x_{i-1}x_i} \oplus g^{x_ix_{i+1}}$, where $\oplus$ denotes the XOR operator. Each participant can compute all DH shared secrets from these broadcasts using knowledge of either their "left" or "right" DH shared secret. Secondly, the authentication mechanism is changed: during the first round, each participant broadcasts g^{x_i} and a digital signature of this value. Moreover, participant n also includes a signed hash of some random "keying bytes" k_n. During the second phase, every participant i—except for participant n—includes their own random keying bytes k_i in their broadcast. Participant n instead includes $k_n \oplus g^{x_nx_1}$ in their second broadcast. Recovering $g^{x_nx_1}$ from the second broadcast in the usual manner is sufficient to derive all keying bytes. The group key becomes $\mathrm{KDF}(k_1 \| k_2 \| \ldots \| k_n)$. The scheme is provably secure in the Random Oracle Model (ROM) with the CDH assumption.

KLL also describe mechanisms to make the group membership dynamic. When a new participant joins the group, they are placed into the logical ring. The protocol then proceeds as normal, except that only three of the old members participate: the left and right neighbors behave normally, but the remainder of the old group is represented by a single participant that uses the previous group key as their secret exponent. All participants still broadcast new keying bytes. Note that this approach naturally supports "mass join" functionality (where more than one participant joins an existing group at once) by adding all new participants to the ring simultaneously. When participants leave the group, their left and right neighbors chose new secret exponents and broadcast their new public keys and XORed DH shared secrets in two rounds.

Only participants that are adjacent to a leaving participant broadcast new keying bytes, and only one of these XORes the bytes with a DH shared secret as in the setup phase. Both the join and leave procedures include digital signatures on the new values for authentication.

Jarecki et al. [JKT07] investigated techniques to make the BD protocol variant from KLL *robust*. Their resulting schemes include redundant messages that can successfully establish a group key without restarting even when some of the participants fail to complete the protocol. Their primary scheme operates by having each participant broadcast XORed DH secrets for multiple possible neighbors. The set of possible neighbors is chosen based on the topology of the T^{th} power[1] of the original ring graph for some failure threshold parameter T. This allows participants to skip failed nodes when computing the group key.

Dutta and Barua [DB08] described the DB protocol, a modification of the original unauthenticated variant of the BD protocol. The design of the DB scheme begins by applying the Katz-Yung compiler [KY03] to the BD protocol for authentication. The authors then manually simplify the resulting construction. The resulting scheme is essentially the original unauthenticated BD protocol with digital signatures incorporating a session identifier for every sent message. The scheme is made dynamic using the same technique as KLL, except without independent keying bytes and with digital signatures on all messages. The primary difference between the DB and KLL protocols is that KLL did not prove the security of their dynamic group operations, and that the full DB protocol is provably secure in the standard model assuming the DDH problem is hard and the digital signature scheme is secure; KLL used the ROM with the CDH assumption.

Abdalla et al. [ACMP10] described a BD variant supporting *subgroup messaging*. This extension to the GKE definition allows participants to efficiently establish shared keys with specific subgroups of the original participant set. A GKE that supports subgroup messaging for 2-party subgroups is called a GKE+P protocol. A GKE that supports subgroup messaging for m-party subgroups (for $1 < m < n$) is called a GKE+S protocol. Abdalla et al. describe two variants of KLL: mBD+P is a GKE+P protocol, and mBD+S is a GKE+S protocol. mBD+P operates similarly to the DB protocol—an application of the Katz-Yung compiler to the BD protocol. The primary difference is that instead of using the DH shared secrets to produce the XORed value during the second round, the DH shared secrets are passed through a key derivation function, KDF_1, first. Subgroup keys are then formed by computing the DH shared secret (using the public keys from the first round) for the participants and passing it through another key derivation function, KDF_2. mBD+S is the same, except establishing subgroup keys requires a single additional round that simply runs the second round protocol again for the subgroup. The security model for the

[1] Given a graph G with vertices V and edges E, the T^{th} power of G is a graph G^T with the same vertices V and a new set of edges E^T such that there is an edge from v_1 to v_2 in E^T if and only if there is a path from v_1 to v_2 of length T or less in G.

protocols requires that subgroup keys are independent from all other subgroup keys and the overall group key across all sessions. Cheng and Ma [CM10] subsequently cryptanalyzed the protocols and found a flaw that requires an additional key confirmation round to correct. The (n+1)sec group messaging protocol described in Section 2.8.5 uses both mBD+P and mBD+S.

Yang and Tan [YT10] presented a modification to the KLL protocol designed to be provably secure with stronger definitions in the standard model with the DDH assumption. The main difference from KLL is that all participants commit to their keying bytes in the first round, and all but the last participant open their commitments in the second round; the last participant still XORs their keying bytes with their "right" shared secret as in KLL. The session identifier is a concatenation of the commitments. The authors define a security model that captures the notions of SK-Security (key secrecy), MA-Security (no honest instance accepts an unmatched instance), and Co-Security (contributiveness; no participant has any control over the value of the group key). They then prove the security of the scheme based on the DDH problem, the security of the digital signature and commitment schemes, and security of two independent pseudo-random function families.

Finally, two extensions to BD were made in 2013. Liu et al. [LVH13] used the original unauthenticated BD scheme to construct a more complex GKE to provide online deniability in GOTR (2013). Li and Xu [LX13] described a minor performance optimization to BD that avoids connecting the first and last participant in the ring (thereby changing the topology to a line). When broadcasting is implemented on a unicast network, this saves two message transmissions at the cost of additional security assumptions; the security relies on the hardness of the "squaring DDH" problem, which asks the adversary to distinguish between (g^x, g^{x^2}) and (g^x, r) for a random group element r.

3.5 Key Trees

By far, the largest area of GKE research relates to *key trees*. Key trees are tree structures where each node is associated with some cryptographic key, and subtrees correspond to subgroups with knowledge of a key. This area of research is substantial because many of the same approaches are useful in other contexts, such as broadcast encryption, multicast communications, and wireless sensor networks, among others. GKEs using key trees form the core of modern secure group messaging protocols. This section surveys some of the more notable tree-based schemes.

Burmester and Desmedt [BD96] adopted many of the techniques from their original BD scheme into a tree-based protocol suite. In the generic form of their new scheme, participants

arrange themselves into a tree by computing the minimum spanning tree of their network graph weighted by some performance measure. One participant acts as the group leader ("conference chair") and becomes the root node of the tree. All participants then perform 2-party key agreements with their neighbors. The root chooses a group key, and sends it in encrypted form to its neighbors using their pairwise keys. Each participant recovers the group key and encrypts it for its descendants using their pairwise keys. The paper then proposes a more sophisticated multicast version [BD96, Algorithm 2] based on Diffie-Hellman exchanges, later referred to as "BD-II" [DLB07]. During the distribution phase, each participant multicasts, to each subtree associated with one of its children, the key it shares with its parent encrypted with the key it shares with the child. For the root, "the key it shares with its parent" is instead the group key. Each participant can then use the key it shares with its parent to decrypt every key up to the root, thereby recovering the group key. Desmedt et al. [DLB07] later added authentication to the BD-II scheme. Simply applying the Katz-Yung compiler [KY03] requires the distribution of $O(n)$ digital signatures, which undermines the logarithmic performance of the scheme. Desmedt et al. showed that modifying the compiler so that signatures are only attached to messages sent during the unauthenticated protocol yields a secure scheme.

In an RFC, Wallner et al. [WHA99] proposed several techniques for establishing cryptographic keys in multicast networks where a server is responsible for key generation and distribution. Their recommended approach is to create a tree where each node contains a symmetric key. The root node's key is the "net key" (i.e., the group key) and is used for group broadcasts. Each participant in the network arranges a shared key with the server, and is associated with a leaf node in the tree. In contrast to the BD-II scheme, participants are not associated with internal nodes in the tree, and the tree does not necessarily correspond to the network architecture. The server then sends to each participant (using their shared key for encryption) the set of all keys on the path from the root to the participant's leaf node. When a participant is removed from the group, all of the nodes on the participant's path must be rekeyed. The server accomplishes this by encrypting the replacement keys with all of the keys for sibling nodes of the participant's path (these nodes are referred to as the *copath*) and multicasting the wrapped keys to the affected participants. This allows every participant who previously knew an affected key, except for the removed participant, to obtain the new values. The primary advantage of this scheme is that key storage and replacement communication costs for each participant is $O(\log n)$ when the tree is balanced.

Wong et al. [WGL00] contemporaneously and independently published work on symmetric key trees. This work and the RFC from Wallner et al. popularized the notion of key trees for GKEs. They define the more general notion of a *key graph*, which consists of a set of participants, a set of keys, and a relation mapping participants to keys they possess. Each participant is associated with a vertex called a u-node. Each key is associated with a vertex called a k-node. Directed

edges defined by the relation always point to a k-node. If and only if there is a path from a u-node to a k-node, then the private key associated with that k-node is known by the participant associated with the u-node. Interesting topologies of key graphs include a star graph, a tree, and a complete graph. The star graph (where every participant shares a single key) is the base case for a GKE; the only reason to add extra keys to the graph is to improve the performance of group management operations in a DGKE. The authors describe a tree-based scheme that is equivalent to the work from Wallner et al. Beyond this, they also describe various approaches for rekeying operations on the tree: user-oriented, where the server multicasts to each subgroup on the copath their set of replacement keys; key-oriented, where each replacement key is subgroup multicasted to its new set of participants; and graph-oriented, where all replaced keys are multicasted to the entire group. The replacement keys are encrypted with the appropriate copath keys in all three cases. In their performance evaluation, the authors provide guidance for selecting the optimal rekeying technique based on the network and group configuration.

Kim et al. [KPT00] were the first to construct a DGKE defined as a key tree that is initialized with DH exchanges. They specifically address guarantees of the underlying network model: their scheme requires View Synchrony (VS) semantics [FLS97]. VS guarantees that everyone sees the same message set between group membership events, the sender's requested message order is preserved, and a message is received by everyone who is in the group from the perspective of the sending application. Kim et al. argue that any GKE either requires VS semantics or must reimplement them. They then describe their fault-tolerant and robust DGKE protocol, Tree-based Group DH (TGDH). TGDH supports mass joins and mass leaves, which the authors refer to as "group merges" and "group partitions". TGDH offers group key secrecy, forward secrecy (with respect to old group keys), post-compromise security (with respect to new group keys), and key independence (which implies the other properties) against passive adversaries. Participants in TGDH are associated with leaf nodes of a binary tree, and each generate DH keypairs. The secret key associated with each internal node is the DH shared secret of an exchange between its two children. Like STR, the TGDH process requires an efficient mapping from group elements to scalars. The group key is the root node's secret key.

Authentication in TGDH is accomplished by ensuring that all messages are signed, timestamped, sequence-numbered, and type-identified. A participant can join a TGDH group with the support of a "sponsor" in the existing group. The new participant broadcasts their DH public key. The new participant and the sponsor form a new internal node in an unambiguous position in the tree. The sponsor computes the new group key and broadcasts all DH public keys that they know. When a participant leaves the group, the rightmost participant in the subtree of the leaving participant's sibling node becomes the sponsor. The sponsor moves their subtree up to replace the parent node of the leaving participant, resulting in a new group key. Mass join and mass

leave events work similarly to an iterative application of these algorithms. The TGDH protocol was refined in a later publication by the same authors [KPT04].

A natural way to improve the performance of DH key trees is to extend the number of children per node. The tripartite Joux protocol [Jou00] can support three children per node. Schemes following this idea appear in publications from both Lee [LKKR03] and Barua et al. [BDS03]. Neither scheme specifies an authentication mechanism. Dutta et al. [DBS04] later incorporated multi-signatures to efficiently defend against active adversaries, but with static group membership. Dutta and Barua [DB05] then described how to achieve authentication in the dynamic setting.

Brecher et al. [BBM09] noted that the TGDH protocol must be restarted if participants fail to complete certain steps. They investigated techniques to add robustness to TGDH so that the remaining participants can still derive a group key even if others fail during the session. They present two schemes: R-TDH1, and IR-TDH1. Both schemes provide several security properties against attackers outside the group: authenticated key exchange, forward secrecy (with respect to earlier sessions), protection against outsider key compromise impersonation attacks, and mutual authentication. IR-TDH1 additionally provides some defense against malicious insiders: protection against insider impersonation attacks, key agreement, resistance against unknown-key share attacks, and contributiveness. The core idea of R-TDH1 is "tree replication". Each participant i builds their own maximally unbalanced full binary tree (as in STR) containing participants i to n with themselves as the leftmost leaf. After the setup round has completed and some of the participants have failed and disconnected, the remaining participants choose the deepest tree with a live node as the leftmost leaf. Any failed participants that remain in the chosen tree are immediately removed with the normal mass leave protocol. IR-TDH1 adds non-interactive zero-knowledge proofs to R-TDH1 to prevent cheating insiders. Unfortunately, Brecher et al. made an error that undermines the correctness of IR-TDH1 in practice; this error is discussed in Section 8.2.9.

Some recent work has focused specifically on GKEs where participants maintain state between protocol sessions (stateful GKEs or stGKEs). Yang et al. [YLL+17] survey multiple security models for authenticated GKEs and propose a new model for authenticated stGKEs. They present a new protocol, TrAGKE, designed to defend against key compromise impersonation attacks, PKI-related chosen identity and public key attacks, forward secrecy attacks, and leakage attacks on ephemeral secret keys. TrAGKE is a dynamic variant of a Joux-based key tree. Chen and Tzeng [CT17] examined the problem of rekeying in situations where participants may miss rekeying rounds. One way to handle this is to cache the rekeying message history and send the cached messages when the participant comes back online, but the participant then needs to process all of the missed rounds in order to derive the current keys. Chen and Tzeng show how to design a tree-based scheme where only the most recent message needs to be processed; rekeying within a group of n

users requires only $O(\log n)$ time and communication size regardless of the number of missed rekeying messages.

3.6 The ASGKA Family

GKEs and DGKEs are defined to output a secret key that is shared by all participants. Wu et al. [WMS+09] considered a related class of protocols that instead derive a shared public key for the group, and distinct individual secret keys for each of the group's members. A protocol of this type is called an asymmetric group key agreement (ASGKA). The motivation for ASGKAs is to enable anyone, including outsiders, to efficiently send secure messages to a group. An ASGKA can also be used to build a broadcast encryption scheme without a trusted dealer. Wu et al. propose a construction with $O(1)$ sized ciphertexts and keys, but $O(n)$ sized broadcast messages during the protocol setup. The construction relies on an uncommon cryptographic primitive called an aggregatable signature-based broadcast. The scheme can uniquely identify misbehaving participants. The scheme is constructed with cryptographic pairings, and its security is based on the bilinear DH exponentiation problem.

Zhang et al. [ZWQD10] extended the previous work from Wu et al. with the goal of building a broadcast encryption scheme. Their newer scheme uses IBE for the initial setup phase so that the group members do not need to distribute public keys before the ASGKA. A trusted entity is needed to act as the PKG for the IBE scheme, but such an entity normally exists in a broadcast encryption setting.

Wu et al. [WQZ+11] attempted to unify the notions of broadcast encryption and ASGKAs with a new primitive called contributory broadcast encryption (CBE). In the context of ASGKAs, a CBE protocol allows a group to generate a group public key and individual group member private keys, but also allows senders to encrypt messages to specific subgroups on an ad-hoc basis. This means that a CBE scheme essentially operates like broadcast encryption without a trusted third party. The initial CBE proposal was only for static groups (with ad-hoc subgroups), but followup work by Phan et al. [PPS12] combined parallel BD sessions and Cramer-Shoup encryption to construct a dynamic variant. All of these schemes require pairing-based cryptography or obscure primitives and hardness assumptions.

3.7 Chapter Summary

This chapter surveyed several classes of GKEs, which are cryptographic primitives that allow a group of participants to efficiently and securely derive a shared secret key. These schemes are of particular importance to secure group messaging because they can be used to build protocols that scale far beyond what is possible with pairwise two-party AKEs. Section 3.1 briefly discussed NIKEs: non-interactive key exchanges that would trivialize secure group messaging; unfortunately, no practical constructions are known. Section 3.2 surveyed GKE schemes that are considered to be inferior to current approaches (typically because they were published when the field was relatively young); these schemes are highly unusual and are worth occasionally re-considering as new cryptographic tools are discovered. Section 3.3 surveyed the CLIQUES family of GKEs, a formerly popular but still interesting group of GKEs that have since been made obsolete by competing protocols with superior performance. Section 3.4 discussed the BD family of GKEs, which use a clever trick to establish a shared key in a small constant number of rounds, unlike prior approaches; the small number of rounds that is independent of the group size makes these GKEs very attractive for interactive key exchanges over the Internet, where round-trip times are the dominant performance cost. This section provides important background for Part III, since the new secure group messaging protocol therein includes a new sub-protocol in the BD family (presented in Chapter 9). Section 3.5 surveyed the most notable GKEs built using "key trees"; this is the family of GKEs that forms the core of modern non-interactive secure group messaging systems, including MLS and the new design in Part III, and thus is critically important background information for understanding recent advances. Finally, Section 3.6 discussed an interesting class of protocols called ASGKAs, which are a type of GKEs in the asymmetric setting; these protocols are one method of allowing outsiders to securely send messages to a group.

While secure group messaging protocols have historically used GKEs to implement their conversation security layer, more recent protocols have asynchronicity and dynamic membership requirements that necessitate a slightly different primitive. Chapter 4 discusses these modern secure group messaging protocols and the new primitive that they use to provide conversation security.

CHAPTER 4 | MLS and Continuous Group Key Agreements

In this chapter: Background

As secure group messaging schemes continued to evolve, it became apparent that the GKE and DGKE schemes discussed in Chapter 3 were no longer sufficient and needed to be augmented. Contemporary secure group messaging schemes like MLS have four specific requirements that cannot be satisfied using a GKE alone:

- Groups must have dynamic membership, with shared keys being securely replaced whenever the membership changes. This necessitates the functionality of a DGKE rather than a simple GKE.
- The protocol must provide forward secrecy and post-compromise security. These properties are achieved by periodically replacing shared secrets using newly transmitted key material. This "key ratcheting" approach prevents old key material from being compromised, and allows compromised key material to be securely replaced. This should occur even in the absence of membership changes.
- The protocol should be "non-interactive" in the sense that all operations can be performed unilaterally (although an always-available server may be involved) without feedback from other group participants. In other words, no operation should require two group members to be online at the same time. This enables mobile use cases where network connectivity is intermittent.
- Group initialization is also non-interactive in order to support use cases like email, where it is possible to specify a recipient list without consent from the recipients. The process of initializing a group produces a "welcome message" containing ciphertexts that enable recipients to recover the shared secret key for the group. This allows the group initiator to send messages immediately, without waiting for acknowledgment that others would like to join the group.

A protocol that provides all of these features is called an Asynchronous Continuous Group Key Agreement (CGKA). Cohn-Gordon et al. [CCG+18] were the first to sketch a protocol that provided all of these features in order to implement a secure group messaging scheme. This protocol sketch was missing some important details, such as the method for providing dynamic membership. Nonetheless, it can be credited as the impetus for the MLS effort. Section 4.1 discusses the design of this pioneering protocol, and Section 4.2 describes its successor, which eventually formed the foundation of MLS. Section 4.3 describes how MLS constructs a complete secure group messaging system from a CGKA. It was not until 2019 that Alwen et al. [ACC+19] recognized the core functionality of these protocols and defined them as CGKAs; Section 4.4 presents their formal definition. Section 4.5 surveys the most recent improvements to CGKA designs, which have not yet been incorporated into MLS.

4.1 ART

Cohn-Gordon et al. [CCG+18] were the first to publish a GKE specifically designed for modern secure group messaging applications. They sketched a protocol called Asynchronous Ratcheting Trees (ART) that provides key secrecy, authentication, forward secrecy, post-compromise security, and non-interactivity. However, ART is an incomplete protocol that is missing multiple important components for a complete secure group messaging solution; most notably, it has no defined mechanism for dynamic group membership. It is possible to add these missing elements using techniques from the literature discussed in Chapter 3, but doing so is non-trivial: the interaction of components often exposes new security or performance problems.

ART is heavily based on TGDH, as discussed in Section 3.5. Like TGDH, the core of ART is a DH-based key tree with each group member represented by a leaf u-node. Figure 4.1 illustrates the key tree for an ART group with exactly four group members: u_1, u_2, u_3, and u_4. Each rectangular box represents a k-node with an associated public key (in a group with generator g suitable for DH exchanges, written in exponential notation); private keys are the values in the exponent and are not written independently for clarity of the presentation. Each circle represents a u-node with an associated group member. If and only if there is a path in the directed graph from a u-node to a k-node, then the member associated with the u-node knows the private key associated with the k-node. In a key tree, the set of nodes connecting the u-node to the root is called the *path* for that group member. The set of sibling nodes is called the *copath*: for u_B in Figure 4.1, the copath would consist of the two nodes associated with g^A and $g^{\mathrm{KDF}(g^{CD})}$. This visual key graph notation is used throughout this dissertation.

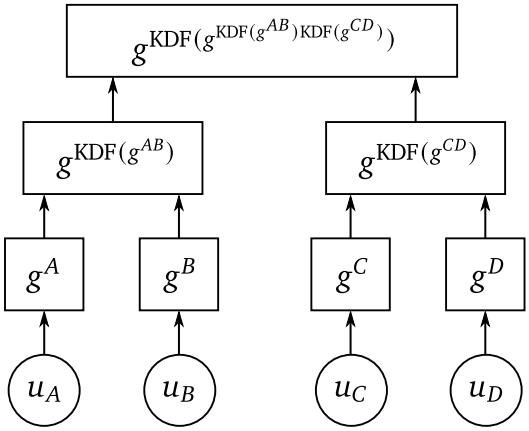


Figure 4.1 AN ART KEY TREE. The group contains four members. The rectangles are *k*-nodes and the circles are *u*-nodes. The values inside the *k*-nodes are their associated *public* keys. (Refs: 55 [a] [b] and 57)

Each private key in the ART key tree is derived by performing a DH exchange between the two children, and then using a KDF to convert the shared secret into a scalar. The main problem when using a TGDH-based scheme like this in a secure group messaging setting arises when non-interactivity is desired: performing a DH exchange to derive the value for a *k*-node seemingly requires interaction between members from the two child sub-trees. ART avoids this problem using two tricks: the group initiator is permitted to learn some private keys that it is not supposed to know, and the group membership is fixed (i.e., ART does not define or use a DGKE).

To enable a non-interactive group setup, ART borrows the notion of "prekeys" from Signal, as discussed in Section 2.7. Each user u_i in the system (not just the group) publishes a long-term identity public key g^{IK_i} and a prekey g^{EK_i}. This prekey can be frequently replaced, as in Signal: users might upload multiple prekeys (one for each group), replace them automatically at regular intervals, or employ other strategies. When an initiator u_A wants to form a group with recipients u_B, u_C, and u_D, they first download a prekey and the identity key for each recipient. The initiator then completes an AKE with each recipient. ART does not specify a particular AKE, but relies upon it for authentication. A reasonable choice for instantiation would be something like 3DH as originally used in Signal. When using 3DH in the example scenario, the initiator would first generate an ephemeral private key e, and then complete the following key exchanges:

$$B = \mathrm{KeyExchange}(\mathrm{IK}_A, e, \mathrm{IK}_B, \mathrm{EK}_B) = \mathrm{KDF}(g^{\mathrm{IK}_A \cdot \mathrm{EK}_B} \| g^{e \cdot \mathrm{IK}_B} \| g^{e \cdot \mathrm{EK}_B})$$
$$C = \mathrm{KeyExchange}(\mathrm{IK}_A, e, \mathrm{IK}_C, \mathrm{EK}_C) = \mathrm{KDF}(g^{\mathrm{IK}_A \cdot \mathrm{EK}_C} \| g^{e \cdot \mathrm{IK}_C} \| g^{e \cdot \mathrm{EK}_C})$$
$$D = \mathrm{KeyExchange}(\mathrm{IK}_A, e, \mathrm{IK}_D, \mathrm{EK}_D) = \mathrm{KDF}(g^{\mathrm{IK}_A \cdot \mathrm{EK}_D} \| g^{e \cdot \mathrm{IK}_D} \| g^{e \cdot \mathrm{EK}_D})$$

Once the initiator generates a private key A for their own leaf node, they now have access to every leaf node private key in the tree (A, B, C, and D). Since they know these keys, the initiator can derive the keypairs for all k-nodes shown in Figure 4.1. When the initiator sends g^e to a recipient, they are able to download IK_A from the key directory and complete the AKE to recover the private key associated with their u-node. To learn all of the keys on their path, a recipient will need to be given, at a minimum, the public keys of k-nodes on the copath. The initiator in ART also sends a MAC of this data to each recipient when initializing the group using a MAC key derived from the AKE with that recipient. The group key is derived from the private key for the root k-node. Since all of these operations can be performed by the initiator (and cached by the server) using only the recipients' prekeys, the entire process is non-interactive and the initiator can immediately send messages encrypted with the group key.

Aside from the setup phase, ART also specifies a key update protocol. Updating the keys provides forward secrecy and post-compromise security. It is also particularly important for members to perform a key update operation after joining a group in order to replace their key material with private keys unknown to the group initiator. During a key update, a member replaces the keys for every k-node on their path. The operation is conceptually simple: the member generates a new keypair for the k-node immediately connected to their u-node, and then derives the new keys on their path using DH exchanges with nodes on the copath. To complete the key update, the member publishes the new public keys on the path along with a MAC derived from the previous group key.

The ART publication does not formally specify a mechanism for dynamic group membership, which would present several problems for this otherwise simple scheme. Notably, the fact that the group initiator initially learns all of the private keys in the tree presents a serious problem if the initiator is removed from the group before other members have performed their first key update, since the initiator will know private keys other than those indicated by the ostensible key tree. Additionally, it is unclear exactly how a group member would add or remove another member unless their leaf nodes are siblings—otherwise, they would need to complete DH exchanges outside of their path. These gaps, among others, inspired the subsequent work described in this chapter.

4.2 TreeKEM

Bhargavan et al. [BBR18] developed a system called TreeKEM that overcomes the limitations of ART. Unlike ART, TreeKEM is an almost fully specified DGKE scheme that supports all of the features required to build modern secure messaging protocols. TreeKEM forms the core of the

MLS specification for this reason. However, the core TreeKEM definition is an abstract protocol and not a wire protocol; as such, it does not specify what cryptographic primitives to use and it provides multiple choices for certain core algorithms. The authors' intention is to keep TreeKEM relatively generic and thus widely applicable, leaving concrete instantiation choices to MLS.

Like ART and TGDH before it, TreeKEM uses a binary tree structure for its key graph. Unlike those protocols, TreeKEM abandons the requirement that k-node keys are derived from DH exchanges. Relaxing this restriction makes it possible to implement simple and efficient group dynamism, and also provides some concurrency benefits.

Each k-node in TreeKEM is associated with an asymmetric keypair. This keypair is generated from an underlying symmetric key, which itself is derived from *exactly one* child node's symmetric key using a cryptographic hash H. The asymmetric keypairs are used by an underlying cryptosystem that is specified as part of the protocol construction.[1] In order to be usable in more applications, TreeKEM allows developers to configure the cryptosystem to use, as well as the function that is used to derive asymmetric key pairs from shared secrets.[2] Figure 4.2 depicts a TreeKEM key tree for a group with four members: u_A, u_B, u_C, and u_D. Note that by convention, ART key trees are normally depicted with public keys in the k-nodes (to emphasize the DH-based structure of the keys), whereas TreeKEM key trees are normally depicted with the underlying symmetric keys in the k-nodes (since the method to derive public keys can be configured based on the application). The internal k-nodes in a TreeKEM key tree are initially derived from their right child using H. The group key can be derived from the root key using a KDF. The group is initialized by u_A in the same way as ART: the initial private keys for recipients (B, C, and D) are separately encrypted to the associated members using an AKE, along with the public keys for subtrees outside of each recipient's path.

The core algorithm in TreeKEM is called a "key update" operation. When a group member performs a key update operation, they begin by generating a new keypair for their leaf k-node. They then compute new keys for each k-node on their path from the bottom up: each key is derived from the (unique) child key that is also on the path using H. For each internal k-node on the path, the new private key is encrypted to the (unique) child key on the copath, and the new public key is broadcast to the whole group (or only to members in its sibling subtree as an optimization). Figure 4.3 depicts the tree from the previous example after two key update operations: u_C performs a key update with a new private key C', and then u_B performs a key update with a new private key B'. Since a key update is effectively a ratcheting operation that

[1] ^ The underlying scheme must be a public-key cryptosystem rather than a KEM, because it must be used to encrypt pre-determined symmetric keys derived using H.

[2] ^ In practice, TreeKEM is typically used with a DH-based public-key cryptosystem. To derive the private key for a k-node, the shared secret is input into a KDF, and the output is interpreted as an integer modulo the group order. The public key for the k-node is then derived as in the normal key generation procedure.

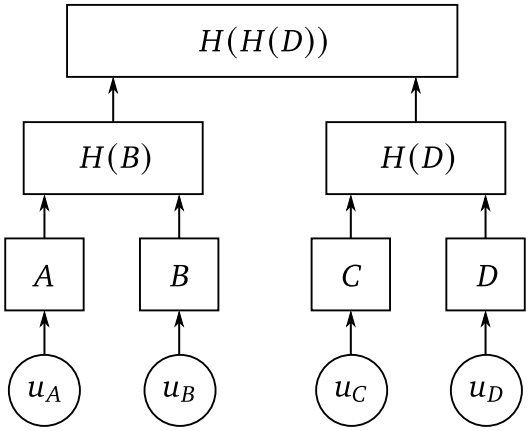


Figure 4.2 A TREEKEM KEY TREE. The group was initialized with four members. The rectangles are k-nodes and the circles are u-nodes. The values inside the k-nodes are the symmetric keys used to derive their associated keypairs. (Refs: 58 and 59)

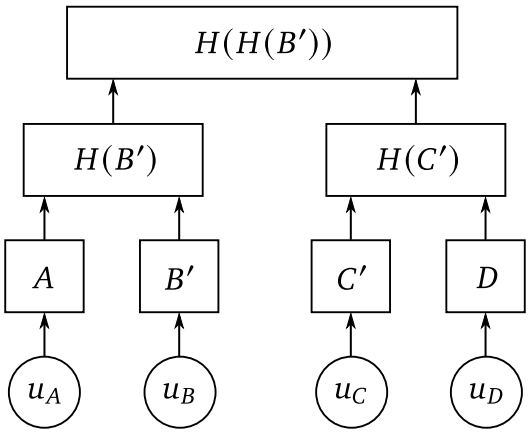


Figure 4.3 A TREEKEM KEY TREE AFTER TWO UPDATES. This tree is derived from Figure 4.2 after a key update from u_C followed by another key update from u_B. (Refs: 58 and 60)

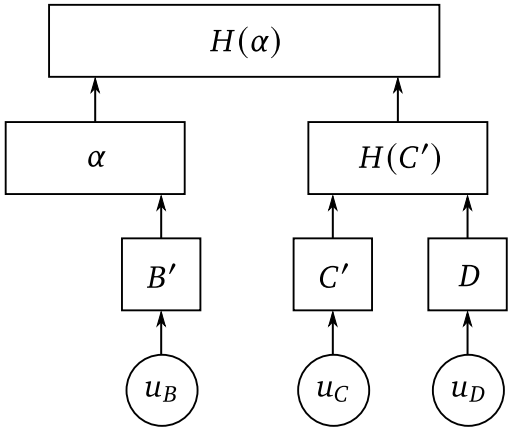


Figure 4.4 A TREEKEM KEY TREE AFTER A REMOVAL. This tree is derived from Figure 4.3 after a group-initiated removal of u_A. (Ref: 60)

replaces all keys known to a member, it provides forward secrecy (because the previous group key cannot be derived from the new root key state) and post-compromise security (because all previously known private keys are replaced by new key material).

TreeKEM's support for dynamic group membership is derived from the key update operation. To non-interactively add a new member to the group, another group member adds new nodes to the key tree and then performs a key update as if they were the new member. The private key for the new member's leaf k-node can be computed and delivered using an AKE in the same manner as group initialization (in fact, group initialization can be modeled and implemented as a sequence of member additions). To non-interactively remove (a.k.a. evict) someone from the group, another group member can perform a key update as if they were the removed member, and then simply remove the deleted member's leaf k-node from the graph instead of generating a private key and delivering it to them. Figure 4.4 depicts the key tree from the ongoing example after another group member has evicted u_A from the group, using a new private key α for the parent of u_A's leaf k-node.

In summary, the original definition of TreeKEM specifies five operations: creating a group, adding someone to a group, evicting someone from a group, performing a key update, and encrypting a message to the group. More precisely, it defines a matching "send" and "receive" function for each of these five operations. Note that this definition of TreeKEM has an inherent confidentiality problem due to the "double join" problem: as part of the group creation, member addition, and eviction operations, the members performing these operations may learn private keys in the tree that they are not meant to know. For this reason, TreeKEM trees are technically not "key graphs" in the sense of Wong et al. [WGL00], because they do not fully capture the

knowledge of keys—they capture the *intended* knowledge. The consequence of this behavior is that evictions may fail to preserve confidentiality: for example, if the group initiator is evicted before other members perform an initial key update, they will still know some remaining private keys and will be able to compute the group key after their "eviction". In general, the way to solve this problem is to keep track of the true key graph and account for all keys known by evicted members. The core TreeKEM protocol does not prescribe a specific algorithm for this task.

In general, TreeKEM provides two main efficiency optimizations: group members may only need to store keys on their own path and copath (instead of the whole tree), and group members may perform certain update operations concurrently. Both of these optimizations negatively impact the security properties of the scheme [BBR18], but this trade-off may be worthwhile for some applications. The TreeKEM authors carefully describe how to handle overlapping update operations when concurrency is enabled, and characterize the security implications of the optimizations.

4.3 MLS

The MLS project is an in-progress IETF collaboration to produce a complete and general purpose secure group messaging wire protocol [BBM+20]. At the core of MLS is a modified instantiation of TreeKEM, as discussed in Section 4.2. Each group member maintains a local group state containing the complete TreeKEM tree (i.e., the partial storage optimization suggested by TreeKEM is not used), the private keys for some k-nodes, and additional group metadata.

The TreeKEM key tree in MLS is always a left-balanced binary tree that preserves the "tree invariant": if a member knows the private key associated with a k-node, then there is a path in the tree from the member's u-node to the k-node (i.e., the member is a descendant of the k-node). Note that this does not imply that the private key for a k-node is known to all descendants—only a subset of the descendants may actually know the private key. For this reason, just like the trees in TreeKEM, the tree in MLS is not technically a key graph as defined by Wong et al. [WGL00]. Each k-node in MLS is associated with a set of "unmerged leaves": the set of u-nodes in the subtree rooted at the k-node that do not actually know the associated private key. In other words, if and only if all sets of "unmerged leaves" are $\emptyset$, then the MLS tree is a key graph. In addition, some k-nodes in an MLS tree may be "blank nodes". A blank node is a k-node with no associated keypair. When an algorithm would normally call for encrypting a key to a blank node, it must instead encrypt the key to the minimal set of k-nodes that covers the same set of descendant u-nodes—this is exactly equivalent to the set of shallowest non-blank k-nodes in the subtree. When encrypting to any k-node (whether blank or not) with a non-empty set of unmerged leaves,

the new key must also be encrypted to each unmerged leaf. By introducing unmerged leaves and blank nodes, MLS enables efficient and secure implementation of eviction operations.

The group state in MLS can be modified by three possible operations: "add", "remove", and "update". These operations are analogous to the same operations in TreeKEM. However, whereas the definition of TreeKEM in Section 4.2 implements each operation as a pair of "send" and "receive" functions, MLS defines a single "proposal" function for each of the three operations, and one global "commit" function that applies a sequence of proposals. Each proposal operation includes some new key material generated by the caller of the function, but does not actually alter the group state. The commit function takes as input a specific sequence of proposals, then outputs a cryptographic message. This message is then provided as input to a "process" function that alters the group state to apply the sequence of proposals given to the commit function. It is important to note that the proposals may be produced by different group members than the member that executes the commit function; the process function is then called by all group members, including the committer. Moreover, "add" and "remove" proposals may actually come from outside of the group in some applications (e.g., when an outside party is responsible for fully or partially managing group membership). MLS encourages an implementation to apply a consistent total order to all commits, ideally using a server-side mechanism [BBM+20, §12].

As in TreeKEM and ART before it, MLS supports non-interactive creation of groups using the notion of "prekeys" borrowed from Signal. In MLS, the combination of a public key with metadata (e.g., supported versions and ciphersuites) is called a "key package". Each key package is used to join exactly one group, with the possible exception of a "key package of last resort"—this is similar to Signal's use of prekeys. Unlike the prior protocols, MLS uses the public key from a key package directly in the key tree, rather than performing an AKE that causes the existing group member to learn the resulting private key.

At a high level of abstraction, the operations in MLS are defined as follows:

1. **Add**: adds a new member to an existing group. The proposal for a member addition consists of a key package previously uploaded by the new member. When committing the proposal, the committer computes a new position in the tree that maintains its left-balanced binary structure, then adds a new k-node and u-node to this position. If necessary, the tree is extended to the right by adding a new root node and an appropriate number of blank nodes. The public key for the new k-node is set to the public key in the key package from the proposal. The new k-node is added to the set of unmerged leaves for every non-blank node on the new member's path. The committer then encrypts the private keys for deepest common ancestor of the new member and committer, as well as every non-blank k-node on the path to that common ancestor, and sends these ciphertexts to the new member as part of a "welcome message". The new member can decrypt these keys to recover the new group state.

2. **Remove**: removes an existing member from the group. When processing a committed remove proposal, the removed member's u-node and the attached k-node are deleted from the tree, and then all intermediate nodes on the path are blanked.

3. **Update**: replaces an existing member's leaf key. This mechanism provides forward secrecy and post-compromise security. When proposing an update, the proposer generates a fresh key pair and includes the public key in the proposal. When processing the operation, the proposer's k-node is updated to use the new public key. All intermediate nodes on the proposer's path are blanked.

As part of a commit, the committer generates new key pairs for every k-node on their own path and encrypts the new private keys to other members in each subtree. This operation works in the same manner as the TreeKEM update described in Section 4.2 (in particular, key pairs are derived from the updated child using a KDF), except that it must take unmerged leaves and blank nodes into account. As a result of this commit operation, any blank nodes on the committer's path become regular k-nodes with associated keys—this is the only mechanism to "unblank" nodes. Moreover, the committer's u-node is removed from any unmerged leaf sets on their path, since they now have access to the private keys for all of the k-nodes that they should.

Group initialization in MLS is equivalent to forming a single member tree (containing only the creating member), proposing an add operation for each recipient, committing these proposals, processing the commit, and then sending welcome messages to the recipients. Figure 4.5 depicts the initial state of the tree in an MLS group after member u_A creates a group with three other members. The public keys associated with B, C, and D are extracted from key packages downloaded from a key directory by u_A when creating the group—these keys were previously uploaded by the recipients. Initially, the three other leaves are all unmerged. When committing the initial group state, u_A generates welcome messages for the recipients: it sends an encryption of $H(A)$ to u_B and encryptions of $H(H(A))$ to u_C and u_D. Figure 4.6 depicts the same example after two more operations have been performed. First, u_B proposes and commits an update for itself: it generates fresh key material B' to derive a new key pair for its k-node, and updates the tree appropriately. The parent k-node is derived from $H(B')$, and the root is derived from $H(H(B'))$. u_B encrypts $H(B')$ to A as part of the commit. This process also removes u_B from the sets of unmerged leaves. Next, u_C proposes and commits a removal of u_A from the group. First, all k-nodes on the path to u_A are blanked, and the u-node and associated k-node for u_A are removed from the tree. u_C then generates fresh key material C', populates the previously blank parent k-node with a key pair derived from $H(C')$, and derives the new root from $H(H(C'))$. This commit involves encrypting $H(C')$ to D and encrypting $H(H(C'))$ to B' (this latter encryption target is derived by looking at the children of the newly blanked k-node). u_C is also removed from the sets of unmerged leaves as a result of performing this commit.

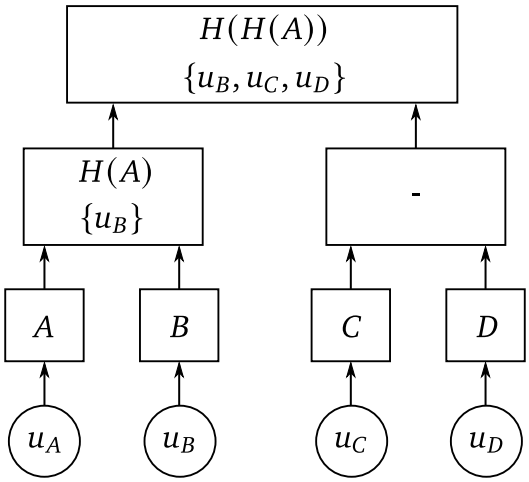


Figure 4.5 AN MLS KEY TREE. The group associated with this tree was created by u_A with other members u_B, u_C, and u_D. Internal k-nodes are depicted with their symmetric key (used to derive the associated asymmetric key pair based on the configured cryptosystem) and their set of unmerged leaves. The k-node containing "-" is a blank node. (Refs: 63 and 64)

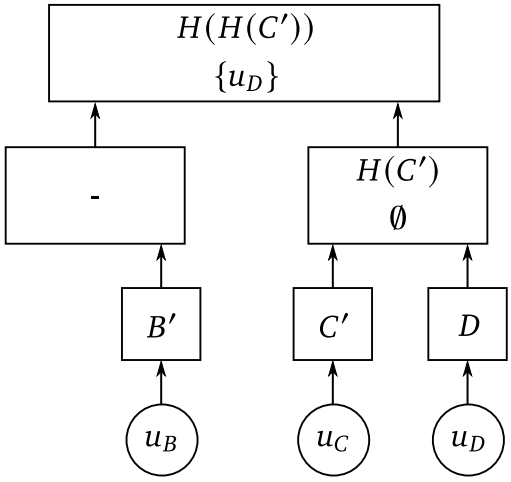


Figure 4.6 AN MLS KEY TREE AFTER AN UPDATE AND A REMOVAL. This tree is derived from Figure 4.5 after u_B commits an update proposal for itself and u_C commits a removal of u_A. (Ref: 63)

4.4 CGKAs

Alwen et al. [ACC+19] were the first to identify that the core algorithms of ART, TreeKEM, and MLS are all variants of a new type of cryptographic primitive: an extension of the DGKE that incorporates non-interactivity into the design. They named this new primitive an Asynchronous Continuous Group Key Agreement (CGKA) [ACC+19], and in 2020 Alwen et al. [ACJM20] presented a formal definition for the primitive. A CGKA is a scheme that implements the following functions, where the protocol state γ is local to each member:

- **CGKA.KeyGen**$() \rightarrow (\text{pk}, \text{sk})$: Generates a new long-term asymmetric key pair (pk, sk).
- **CGKA.Create**$() \rightarrow (\gamma, e)$: Returns a fresh protocol state γ representing a group with the party running the algorithm as the sole participant. e is the identifier for the initial *epoch*.
- **CGKA.ProposeAdd**$(\gamma, \text{id}_t, \text{pk}_t) \rightarrow (\gamma', p)$: Proposes adding a member with identifier id_t and long-term public key pk_t to the group. Takes the protocol state γ as input, and outputs a new state γ' and a proposal p.
- **CGKA.ProposeRemove**$(\gamma, \text{id}_t) \rightarrow (\gamma', p)$: Proposes removing a member identified by id_t from the group. Takes the protocol state γ as input, and outputs a new state γ' and a proposal p.
- **CGKA.ProposeUpdate**$(\gamma) \rightarrow (\gamma', p)$: Proposes updating the calling member's key material. Takes the protocol state γ as input, and outputs a new state γ' and a proposal p.
- **CGKA.Commit**$(\gamma, \vec{p}) \rightarrow (\gamma', c, w)$: Applies (a.k.a. *commits*) the proposal messages in vector $\vec{p}$ to the group. Takes a protocol state γ as input, and outputs a new state γ', a commit message c, and a welcome message w. The welcome message may be the empty symbol $\perp$ if $\vec{p}$ does not contain any proposed additions to the group.
- **CGKA.Join**$(\text{sk}, w) \rightarrow (\gamma', \overrightarrow{\text{id}}, \text{id}_i)$: Joins a group using the long-term private key sk and the welcome message w. Outputs a protocol state γ', a list of existing group members $\overrightarrow{\text{id}}$ (where each vector element is the identifier for an existing member), and the identifier id_i for the member that invited the caller to the group (i.e., the one that created w).
- **CGKA.Process**$(\gamma, c, \vec{p}) \rightarrow (\gamma', e', \overrightarrow{\text{info}})$: Takes as input a protocol state γ, a commit message c, and the corresponding list of proposal messages $\vec{p}$. Outputs a new protocol state γ' having epoch identifier e' such that all of the valid proposed changes in the sequence $\vec{p}$ have been

applied to the state. The output $\overrightarrow{\text{info}}$ describes the changes to the group that have occurred while processing the commit message; it has the form $(\text{id}, \overrightarrow{\text{propSem}})$, where id is the identifier for the member that produced the commit message, and $\overrightarrow{\text{propSem}}$ is a sequence of semantic group changes. Each element of $\overrightarrow{\text{propSem}}$ has the form $(\text{id}_s, \text{op}, \text{id}_t)$, where id_s is the identifier for the member that proposed the change, $\text{op} \in \{\text{“addP”}, \text{“remP”}\}$ denotes the type of change, and id_t is the identifier for the member targeted by the change (i.e., either added to or removed from the group).

- **CGKA.Key**$(\gamma) \to (\gamma', K)$: Takes as input the protocol state γ. Outputs the shared secret key K for the group, and a new protocol state γ' in which the key has been erased (for forward secrecy purposes). If the key has already been erased from γ, this function outputs the special symbol $\perp$.

CGKAs are designed to be used to construct a higher-level secure group messaging protocol like MLS. All participants in the system use CGKA.KeyGen to generate long-term key pairs; the public keys are distributed and verified using the trust establishment mechanism. To create a group, an initiator calls CGKA.Create to produce an initial state and epoch identifier. Each participant in a group conversation maintains a local state for the group. Over time, changes to the group are proposed by group members (using CGKA.ProposeAdd, CGKA.ProposeRemove, or CGKA.ProposeUpdate), and then “committed” by a group member (using CGKA.Commit). The act of committing a set of proposed changes causes them to be applied to the group in a specific sequence, producing a state with a new epoch identifier and shared secret key. Committing changes produces a commit message that must be distributed to other group members; all members (including the committer) must call CGKA.Process in order to derive the group state for the new epoch. When a proposed addition to the group is committed, a “welcome message” is generated. When the new group members receive the welcome message, they can use their long-term private key to recover key material from the message, thereby initializing their own local state for the group. The security properties of a CGKA are defined in a simulation-based model—interested readers will find formal security definitions and proofs in the work by Alwen et al. [ACJM20].

Section 4.3 previously described a secure group messaging scheme built from a variant of TreeKEM. This TreeKEM variant implicitly implements the CGKA definition; the CGKA in MLS can be replaced with another system that implements the definition in order to improve the security or efficiency of the overall protocol.

4.5 Improvements to TreeKEM

Several notable works have modified the original TreeKEM [BBR18] scheme discussed in Section 4.2 in order to improve the security properties or apply the system in new settings.

Weidner [Wei19] modified TreeKEM in order to construct a privacy-enhancing collaborative editing system—a system that allows multiple users to edit a shared document simultaneously. These systems require causality preservation of messages, but not necessarily a consistent global transcript. Weidner points out [Wei19, §4.2.3] that TreeKEM's support for concurrent updates undermines post-compromise security in certain scenarios. For example, if the adversary corrupts a group member u_B, members u_B and u_A perform concurrent key updates, and u_B's update is applied before u_A's update after merging, then the adversary will learn the new group key because u_A's update encrypts the new keys to u_B's old (compromised) keys. Since this weakness is considered unacceptable for the collaborative editing setting, Weidner describes a variant called "Causal TreeKEM" that supports concurrent key updates, but exploits the properties of the underlying DH-based group to combine keys (instead of overwriting them) during key updates. This variant provides the post-compromise security guarantees needed by the application. In addition to this contribution, Weidner also defines more granular conversation security properties for secure group messaging in a manner inspired by Unger et al. [UDB+15].

In the same work that initially recognized the CGKA primitive, Alwen et al. [ACC+19] defined a variant called Tainted TreeKEM (TTKEM). This scheme was based on a previous draft of the MLS protocol that did not yet separate the notions of "proposing" and "committing" operations, so the CGKA definition in this work is now outdated. Nonetheless, the main idea of TTKEM remains valuable: instead of using "blank nodes" as described in Section 4.3, TTKEM introduces "tainted nodes". A tainted node is a k-node that is known by members that it should not be; a tainted node is associated with a set of u-nodes with superfluous knowledge of the private key. Instead of blanking a node during an "add" or "remove" operation, the proposer generates a new keypair for the k-node, encrypts the private key to the appropriate subtrees, and adds their own u-node to the k-node's tainted set. When a member performs a commit operation, all k-nodes on their path become untainted. When a key is encrypted to a node with a non-empty tainted set as part of any operation, the k-node associated with the new key material inherits the tainted set. When removing a member, all k-nodes with that member in their tainted set must be updated—not just the removed member's path as in MLS. The TTKEM approach represents a trade-off: more keys must be replaced when evicting members, but more internal k-nodes are available as encryption targets during other operations.

Alwen et al. [ACDT20] identified a weakness in the forward secrecy properties of TreeKEM that allows an adversary to recover previously captured plaintexts after corrupting members in certain scenarios. They proved that this weakness could be robustly overcome by replacing the public-key cryptosystem with "updatable public-key encryption" (UPKE). A UPKE scheme is identical to a PKE scheme, except that the encryption function outputs a new public key in addition to the ciphertext, and the decryption function outputs a corresponding new private key in addition to the recovered plaintext. An important security property of a UPKE scheme is that a private key output by the decryption function cannot decrypt ciphertexts that were previously encrypted to the old private key. The purpose of UPKE is similar to puncturable encryption [GM15], but several trivial and efficient constructions for UPKEs exist. For example, it is simple to construct a UPKE from ElGamal encryption by updating a public key g^x to be $g^x \cdot g^y$, and encrypting y as part of the encryption function.

Most recently, Alwen et al. [ACJM20] presented an approach to achieve insider security in MLS. Their scheme alters TTKEM to include a hierarchical identity-based encryption (HIBE) scheme that ensures all keys are updated with every operation. They also introduce non-interactive zero-knowledge proofs that demonstrate knowledge of symmetric secret keys in certain situations. The new scheme is proven secure in a simulation-based model. The main weakness of the approach is that the zero-knowledge proofs must prove statements about cryptographic hash functions, which typically requires either prohibitively slow proof systems, or very strong security assumptions.

4.6 Chapter Summary

This chapter introduced CGKAs and their application to MLS and other modern secure group messaging protocols. A CGKA scheme is an extension of a DGKE that supports non-interactive group creation and ongoing group key ratcheting. Section 4.1 covered the ART protocol, which was the first published non-interactive secure group messaging framework; it inspired the creation of the MLS working group. Section 4.2 discussed TreeKEM, which was the first true CGKA (although it was not abstracted in those terms at the time) and the core of MLS. Section 4.3 covered MLS itself, which is the most advanced secure group messaging protocol supporting non-interactive environments that currently exists. After the publication of early drafts of MLS, Alwen et al. [ACC+19] recognized that the core primitive in the TreeKEM construction was generalizable; they extracted and defined the CGKA primitive, which is covered in Section 4.4. Finally, Section 4.5 covered several improvements to TreeKEM that have been recently proposed, including TTKEM.

The content that was presented in this chapter and the rest of Part I provides the background necessary to contextualize the contributions of this work. Part II introduces new cryptographic protocols to improve the deniability of two-party secure group messaging schemes; these contributions are expressed using the terminology and context that was introduced in Chapter 2. Chapters 3 and 4 established the foundation for understanding the design of the new secure group messaging protocol introduced in Part III, which uses techniques that are very similar to TTKEM.

PART II

Deniability Against Coercion and Insiders

CHAPTER 5 | Designing DAKEs for Modern Two-Party Secure Messaging

In this chapter: Discussion | Protocol | Implementation | Evaluation

Deniability—the ability to plausibly deny sending a secure message—is a desirable feature for secure messaging tools in certain contexts. As defined in Section 2.4, a "deniable" secure messaging protocol is one in which denying message authorship is just as plausible as it would be for an insecure (i.e., unauthenticated) messaging protocol from the perspective of a "judge". Since most Internet communication protocols have historically lacked cryptographic authentication, deniability can be viewed as the "default" state. Consequently, users may be unpleasantly surprised to discover that the tool they are using lacks deniability. A notable example of this situation occurred in the high-profile data leak of emails from the Clinton campaign in the 2016 US presidential election, where DKIM digital signatures were used to strongly indicate the authenticity of some emails in the leak despite the authors' denials [Ros16].[1] When designing a secure messaging protocol with authentication, it is easy to accidentally enforce non-repudiation of message authorship. For example, digitally signing messages with a long-term identity key provides not just authentication, but also non-repudiation. To avoid mistakenly harming users, a protocol designer should explicitly decide whether to provide deniability or non-repudiation, and make this choice clear. Since deniability is the default state, choosing to introduce non-repudiation has the effect of increasing the power of third-party observers (who gain the ability to cryptographically verify authorship) while potentially decreasing the power of conversation participants (who lose any ability that they may have had to plausibly deny authorship). This may be desirable in some contexts (e.g., when public accountability is more important than candid conversation), but not in others (e.g., casual private messages between friends).

Deniability is difficult to preserve while simultaneously providing authentication. Moreover, the very notion of deniability is difficult to precisely define. Section 2.4 covered several common

[1] ^ The presence of a DKIM signature ensures that the contents of an email and important metadata cannot not be forged without the private key of the signing server. In most cases, this dramatically limits the plausibility of any denial of authorship. Specter et al. [SPG21] recently proposed several new protocols to make DKIM signatures deniable.

definitions of deniability, and Section 2.5 surveyed the history of DAKEs—the primary method of preserving deniability in secure messaging protocols. Ideally, it should be possible to design modern secure messaging protocols that provide strong deniability in addition to authentication; strong deniability is an important form of insider security. As discussed in Section 2.5, three previously introduced protocols ostensibly meet these requirements: Walfish's protocol Φ_{dre} [Wal08], and Unger and Goldberg's protocols RSDAKE and Spawn [UG15]. However, Φ_{dre} is inferior to RSDAKE in terms of performance [UG15], and RSDAKE and Spawn lack some key properties that are desired by the designers of secure messaging protocols like OTR and Signal:

1. the protocols are too slow to use in real-world secure messaging settings like smartphone applications;
2. the protocols cannot provide forward secrecy against active adversaries in non-interactive settings;
3. the protocols do not provide forward secrecy against future quantum adversaries;
4. the protocols make users susceptible to Key Compromise Impersonation (KCI) attacks; and
5. the protocols are not contributory—shared secrets are unilaterally determined by one party, which permits sophisticated attacks against key secrecy.

This chapter presents three new DAKEs—DAKEZ, ZDH, and XZDH—that overcome the aforementioned limitations with RSDAKE and Spawn. These are the first practical mechanisms for achieving strong deniability in two-party secure messaging settings, including non-interactive scenarios.[2] Moreover, these DAKEs can act as drop-in replacements for the DAKEs in protocols like OTR and Signal in order to efficiently provide strong deniability without sacrificing any existing security properties. The protocols also explicitly include the option to add quantum transitional resistance, which preserves the confidentiality of past conversations in the event that future quantum computers are able to break present-day cryptography. While the new protocols are limited to the two-party case, the threat model and design methodology in this chapter provide important and generalizable insights; Chapter 10 draws upon these insights to design a deniable secure group messaging protocol.

The chapter is organized as follows: Section 5.1 motivates the desire for online deniability; Section 5.2 defines the security properties and features of the new DAKEs, and outlines their

[2] ^ The protocols do make a few concessions that weaken the deniability guarantees in some circumstances. Notably, the protocols only provide online deniability for one participant in the non-interactive case. These limitations seem to be inherent to the settings.

design; Section 5.3 establishes notation and introduces constructions for cryptographic primitives; Section 5.4 defines DAKEZ; Section 5.5 defines ZDH; Section 5.6 defines XZDH; Section 5.7 covers practical considerations for deploying the new DAKEs; Section 5.8 describes a prototype implementation of the new DAKEs; Section 5.9 evaluates the time and space efficiency of the protocols and existing key exchanges; and Section 5.10 discusses key compromise impersonation attacks in the context of strongly deniable key exchanges. Security proofs for the schemes are presented in the next chapter, Chapter 6.

The content of this chapter was previously published by Unger and Goldberg [UG18]. Subsequently, the DAKEs described in this chapter were incorporated into the OTRv4 protocol [OTR17], where they serve as the core mechanism for providing deniable authentication. The prototype described in Section 5.8 served as a reference for independent implementations of the abstract protocols. While OTRv4 is not yet deployed, it will soon provide strong deniability for end-user secure messaging tools.

5.1 Online Deniability Attacks

Offline and online deniability defend against an insider attack where one party in the conversation provably leaks messages to an outsider. As discussed in Section 2.4, online deniability is a relatively new consideration for secure messaging protocols, which have traditionally focused on offline deniability. It is natural to wonder if online deniability is truly necessary, or if offline deniability alone is sufficient to defend against the insider attack in practice.

This section demonstrates what can go wrong in a secure messaging protocol that overlooks online deniability by presenting two insider attacks against two popular protocols in great detail. Section 5.1.1 presents an insider attack in which a protocol participant is forcibly coerced by an outsider into proving the authorship of incoming messages. Section 5.1.2 considers a scenario in which a malicious insider reserves the right to selectively prove the authorship of incoming messages to outsiders at a future time. Both sections describe the technical details of implementing these attacks against the OTRv3 [OTR16] and Signal [Ope13] protocols.

These sections assume that the reader is familiar with the technical details of the relevant specifications, including X3DH [MP16] and the double ratchet [PM16] used by Signal; these sections use the notation of the specifications without definition or explanation. Since the purpose of this section is merely to illustrate that online deniability is an important consideration and the technical details do not directly inform the design of defenses, readers that already accept the importance of online deniability may wish to skip to the next section.

5.1.1 Coercive Judges

In this insider attack, an online judge[3] coerces a protocol participant into interactively proving that messages were authored by a victim, without compromising long-term secrets. The judge is able to detect if the coerced participant deviates from the protocol.[4] As in all online deniability attacks, the participant shares a secure channel with the judge. The same general approach works for attacking both OTRv3 and Signal:

1. The AKE is completed between the victim and the judge. The coerced participant provides the authenticating information necessary to cause the victim to believe they are communicating with the participant, rather than the judge.
2. The judge conducts the resulting conversation normally, with no need for the coerced participant other than relaying ciphertexts. The judge can be certain that all messages they exchange are with the victim.

5.1.1.1 Attacking OTRv3

In this example, Judson is the judge, Alice is the victim, and Bob is the coerced participant. Alice and Bob are the protocol participants described in the OTRv3 specification [OTR16].

1. Bob establishes a connection to Alice.
2. Judson picks random values r and x.
3. Judson sends $\text{AES}_r(g^x)$ and $\text{HASH}(g^x)$ to Alice through Bob.
4. Alice replies with g^y.
5. Judson computes M_B normally, forces Bob to produce $\text{sig}_B(M_B)$, and verifies the signature.
6. Judson continues normally and sends r, $\text{AES}_c(X_B)$, and $\text{MAC}_{m2}(\text{AES}_c(X_B))$ to Alice through Bob.

[3] As discussed in Section 2.4, an online judge is an entity that interacts with a protocol participant in order to evaluate if the purported ongoing conversation is genuine.

[4] It is theoretically possible that two online judges cause two coerced participants to implicate each other. This is still a valid online deniability attack, because both judges learn that the conversation was "genuine" in the sense that the protocol was not simulated. The attack does not reveal the *motives* of the participants—only that the conversation between the participants (who may have been coerced by outsiders to send messages) truly occurred.

7. Alice sends $\text{AES}_{c'}(X_A)$ and $\text{MAC}_{m2'}(\text{AES}_{c'}(X_A))$.

8. Judson now shares $s = g^{xy}$ with Alice and continues the OTR session, relaying ciphertexts through Bob.

The reverse attack, where Judson coerces Alice into establishing a session with Bob, is nearly identical.

5.1.1.2 Attacking Signal

In this example, Judson is the judge, Alice is the coerced participant, and Bob is the victim. Alice, Bob, and the server are the entities in the X3DH specification [MP16].

1. Alice contacts the server and receives Bob's prekey bundle containing the identity key IK_B, the signed prekey SPK_B, the signature $\text{Sig}(IK_B, \text{Encode}(SPK_B))$, and (without loss of generality) the one-time prekey OPK_B.

2. Judson generates a key pair with public key EK_A.

3. Judson forces Alice to compute and reveal $DH1 = \text{DH}(IK_A, SPK_B)$. Optionally, Judson may ask for a zero-knowledge proof of correctness for $DH1$.

4. Judson sends EK_A to Bob through Alice.

5. Judson computes $DH2$, $DH3$, and $DH4$ using the secret for EK_A. Judson then computes $SK = \text{KDF}(DH1 \| DH2 \| DH3 \| DH4)$, which is also known by Bob. Judson now continues the Signal session normally, relaying ciphertexts through Alice.

Note that while this attack does not reveal IK_A to Judson, it does slightly weaken the security of future exchanges with IK_A and SPK_B; this is alleviated when SPK_B is replaced or one-time prekeys are used by Bob.

The reverse attack, where Judson coerces Bob into communicating with Alice, is more complex because Judson must generate the prekeys. In that case, the risks to Bob can be mitigated by using a distributed key generation scheme [GJKR99] to jointly generate the prekeys. When Alice selects a bundle, Bob sends the associated shares to Judson, who reconstructs the secrets. For bundles selected by non-victims, Judson sends the associated shares to Bob, who proceeds unmonitored.

5.1.2 Malicious Users

In this insider attack, a malicious participant interacts with a purpose-built third-party service during a conversation with a victim. Assuming that the public keys of the participant, victim, and service are known and validated, the participant and service did not collude to forge the transcript, and the service does not have direct access to the connection for the conversation, the participant is able to produce non-repudiable proof of message authorship by the victim. It is possible to do this while protecting the participant's long-term keys, preventing the service from reading or modifying the conversation, preventing forgery by any individual party or outsider, allowing the participant to selectively disclose only a portion of the conversation, and preventing a forward secrecy breach when only the service is compromised.

The overall attack involves using the service to produce a trustworthy log of the protocol transcript (the exact network data transmitted to and from the participant), and enabling the participant to selectively publish ephemeral keys for ciphertexts to reveal portions of the conversation transcript (the plaintext messages exchanged in the session).

The basic approach is the same for both OTRv3 and Signal: the service maintains the keys for the session and otherwise behaves honestly, while the participant handles sending and receiving ciphertexts. Both the participant and service digitally sign protocol messages and, when necessary, they use secure Multi-Party Computation (MPC) to allow the participant to conduct the conversation. Alternatively, a different trustworthy logging mechanism, like Intel SGX secure enclaves [AGJS13], can be used instead; Gunn et al. [GPA19] detail a generalized form of the attack using remote attestation.

5.1.3 Attack Overview

The protocol begins by performing the attack against the AKE from Section 5.1.1 in such a way that the participant is given proof of the victim's identity and the service cannot select a specific ephemeral secret. When the participant receives a message:

1. The participant signs the protocol message.
2. The participant and service perform secure MPC to verify the authenticity of the protocol message and decrypt the message for the participant (but not the service).
3. The service signs the participant's signature and as much of the protocol message as possible without being given enough data to decrypt it.

When the participant wants to send a message:

1. The participant and service use secure MPC to generate the encrypted and authenticated protocol message.
2. The participant signs the protocol message.
3. The service signs the participant's signature and as much of the protocol message as possible.

While the conversation is ongoing, before erasing ephemeral keys needed to decrypt messages, the service encrypts them under a public key generated by the participant at the start of the protocol, and stores them. When the participant wants to incriminate the victim:

1. The participant asks the service to terminate.
2. The service sends the encrypted ephemeral keys it has stored, and any currently active keys, to the participant. The service then terminates the session.
3. The participant publishes the signatures, proving that the protocol transcript is real and unmodified.
4. The participant can selectively reveal messages by decrypting and publishing the ephemeral keys needed to decrypt and authenticate target messages.

5.1.3.1 OTRv3 Attack Details

When the service performs the attack from Section 5.1.1 on OTRv3, the ephemeral key is produced using distributed key generation [GJKR99] and the participant's share is revealed to the service. The service reveals r, $m1$, c', and $m1'$ to the participant to verify the victim.

When receiving a message, secure MPC is used to check the MAC; the victim keeps the ciphertext and received MAC secret, and the service keeps mk secret. To decrypt, the service verifiably reveals the AES-CTR keystream for the received ctr using secure MPC, and the participant uses it to privately decrypt the ciphertext. The service only assists with decryption after signing the participant's signature on the protocol message. It is not necessary for the service to verify the participant's signature, since cheating will be detected during verification of the incriminating transcript.

When sending a message, the participant generates a symmetric key. The parties then use secure MPC to encrypt and authenticate a message (known to the participant) with the message

key (known to the service), with the resulting ciphertext being encrypted under the participant's symmetric key (so it is not revealed to the service). The service stores encrypted versions of the expired DH secrets that can be revealed during the incrimination procedure.

5.1.3.2 Signal Attack Details

As in the attack against OTRv3, the attack against Signal also begins with producing the ephemeral key using distributed key generation. For Signal, the participant uses the signed prekey to verify the victim. When receiving a Signal message, the service simply reveals the message key to the participant. Sending a message uses the same procedure as the OTRv3 attack: the participant performs secure MPC with the service to encrypt and authenticate the message. Unlike for OTRv3, it is not necessary for the service to store encrypted versions of message keys in the Signal attack, since message keys are always revealed immediately upon receipt of an incoming message.

5.2 Designing Defenses 

The three new protocols, Deniable Authenticated Key Exchange with Zero-knowledge (DAKEZ), Zero-knowledge Diffie-Hellman (ZDH), and Extended Zero-knowledge Diffie-Hellman (XZDH), are meant to be used in common secure messaging scenarios. DAKEZ is designed for use in interactive settings such as instant messaging applications, while ZDH and XZDH are designed to be used in non-interactive settings such as text messaging. ZDH is the most efficient, but XZDH provides stronger forward secrecy than ZDH. The new DAKEs share many design similarities with RSDAKE and Spawn, but they are 4000–5000 times more efficient in practice. This significant efficiency advantage comes from constructing and using cryptographic primitives that rely on the ROM for security proofs.

Section 5.4 constructs DAKEZ by adopting RSDAKE's approach to authentication, but using several new instantiations of cryptographic primitives in the ROM. Additionally, slight improvements in the design of DAKEZ provide some enhanced security properties. For comparison purposes, Section 5.5.1 adapts Spawn to the random oracle model using the new primitives. Examining this ROM-based variant of Spawn reveals that performance can be further improved by leveraging the fact that Spawn provides only partial online deniability; this leads to the development of ZDH, described in Section 5.5.2. Finally, Section 5.6 describes XZDH, a variant of ZDH that adopts X3DH's forward secrecy technique to gain more control over the confidentiality guarantees.

5.2.1 Proof Technique

There are two general approaches to proving the security of key exchanges: *security by indistinguishability*, where security properties are expressed in terms of indistinguishable adversarial games [BR93a]; and *security by emulation*, where a protocol is shown to be indistinguishable from an idealized protocol with access to secure channels and a trusted third party [CK02b]. The new DAKEs are designed to be proven secure using the emulation method through the Generalized Universal Composability (GUC) framework [DKSW09], which was also used to prove the security of RSDAKE and Spawn [UG15]. GUC-based security proofs are generally more convincing of real-world security, more resistant to insecure composition, and able to naturally express deniability properties. On the other hand, GUC-based proofs are often overly restrictive and more complex.

While RSDAKE and Spawn were proven secure using the GUC framework, the proof sketches used the standard model with obscure hardness assumptions. The new protocols can be proven secure using random oracles and widely accepted standard assumptions. While the use of random oracles in security proofs has historically caused debate among theorists [KM15], it remains to be shown that instantiation of random oracles using appropriate cryptographic hash functions in real-world protocols introduces any actual security flaws.

The new DAKEs are designed to provide many security properties and features. However, proof of these properties within the GUC framework requires the definition of an *ideal functionality* and proof showing that the real key exchanges behave indistinguishably from the idealized version. Formal definitions of these ideal functionalities, explanations for how they exhibit desirable properties, and expression of the resulting security theorems, are long and complex. For ease of presentation, the formal definition of the GUC ideal functionalities, security theorems, and proof sketches are deferred to Chapter 6. This chapter focuses on intuitive definitions of the new DAKEs and practical issues surrounding their deployment. While this chapter does not contain the formal proof sketches, to facilitate design discussion and comparison with prior schemes, the next section defines high-level security objectives for the new DAKEs, and simplified security propositions are presented throughout this chapter.

5.2.2 Protocol Properties

The new DAKEs provide the following properties:

1. **Universally composable AKE** [CK02b]: The protocols emulate idealized authenticated key exchanges facilitated by a trusted third party (without requiring one in practice), and continue

to do so even when composed within arbitrary protocols. This property includes the traditional notions of mutual authentication, key secrecy, and key freshness [BR93a; CK02b].

2. **Offline deniability** [DGK06]: Anyone can forge a DAKEZ, ZDH, or XZDH transcript between any two parties using only their long-term public keys. Consequently, no transcript provides evidence of a past key exchange, because it could have been forged. This is similar to the deniability offered by 3DH [Mar13].

3. **Online deniability** [DKSW09]: Participants in a DAKEZ exchange cannot provide proof of participation to third parties without making themselves vulnerable to KCI attacks, even if they perform arbitrary protocols with these third parties during the exchange. ZDH and XZDH provide this property for one party.

4. **Contributiveness**: The new DAKEs are *initiator-resilient* [HMS03] in the same sense as traditional Diffie-Hellman [DH76] key exchanges—the initiator of the protocol cannot force the shared secret to take on a specific value. In practice, it is also computationally infeasible for the responder to select a specific shared secret, although it may attempt to select one with some desirable characteristic through a brute-force search. The initiator (and, in practice, the responder) cannot force the shared secret to be a value known to a third party before the exchange begins.

   Without contributiveness, an adversary can specify a secret, coerce a participant into forcing that shared secret, and then decrypt the subsequent conversation without access to the key exchange transcript [HMS03]. A non-contributory protocol also permits a participant to force a shared secret known to an innocent third party, enabling that third party to decrypt the conversation without their consent.

5. **Forward secrecy** [BPR00]: A classical adversary that compromises the long-term secret keys of both parties cannot retroactively compromise past session keys. DAKEZ offers *strong forward secrecy*—it protects the session key when at least one party completes the exchange. ZDH offers *weak forward secrecy* [BPR00]—it protects the session key only when both parties complete the exchange, but not when only one party completes and begins using the session key. XZDH protects completed sessions and incomplete sessions that stall long enough to be invalidated by a participant. Section 5.6 discusses this distinction in greater detail.

6. **(Optional) quantum resistance** [SWZ16]: The new DAKEs can optionally be combined with a quantum-resistant KEM to maintain forward secrecy against future quantum adversaries, at the expense of performance.

7. **Post-specified peer** [CK02a]: Participants in DAKEZ exchanges learn the identities of their partners during the exchange. For ZDH and XZDH exchanges, only one party begins the protocol with knowledge of the other's claimed identity.

In all configurations, the new protocols are universally composable AKEs with offline deniability, contributiveness, and post-specified peers. Given these baseline properties, this chapter describes several configurations that offer different tradeoffs between online deniability, forms of forward secrecy, computational efficiency,[5] and quantum resistance; the best configuration to use depends on the particular secure messaging context.

5.2.3 Quantum Transitional Security

The new DAKEs include a mechanism to address the problem of forward secrecy against future quantum adversaries. The purpose of forward secrecy in key exchange protocols is to prevent future adversaries from retroactively compromising recorded sessions. This property is extremely important because it minimizes the impact of compromised long-term keys. However, current deployments of key exchanges with forward secrecy are believed to be vulnerable to quantum cryptanalysis. This leads to the worrying possibility that all communications that are currently protected by "secure" protocols like Signal might be retroactively decrypted by a passive adversary that gains access to a quantum computer in the future; such an event could be potentially devastating to many real users. As quantum computation continues to appear more feasible, it is becoming increasingly clear that defenses against this potential threat must be deployed as quickly as possible. However, implementing protocols that are fully resistant to all quantum attacks remains challenging and expensive. Moreover, the security (even classically) of "quantum-resistant" cryptosystems remains suspect, relative to the confidence in classical schemes based on widely examined hardness assumptions.

An interesting class of key exchanges and KEMs are "*hybrid*" schemes that provide *quantum transitional security*. Bindel et al. [BBF+19] define security notions for four security settings based on when quantum computers become capable of breaking the cryptosystems and whether or not the communication protocol itself is quantum:

- C^cC: the "classical" setting. In this setting, the protocol is classical and the adversary never gains access to quantum computation.

[5] The constructions are preferentially optimized based on computational efficiency rather than other performance metrics, such as network transmission size.

- C^cQ: the "future quantum" setting. In this setting, the protocol is classical and the adversary gains access to quantum computation at some point in the future (after the protocol sessions have completed).
- Q^cQ: the "post-quantum" setting. In this setting, the protocol is classical and, while the adversary already has access to quantum computation, the adversary is restricted to interacting with the protocol classically.
- Q^qQ: the "fully quantum" setting. In this setting, the adversary and the parties performing the protocol are all able to perform quantum computations and send quantum communications.

Typically, "quantum transitional security" refers to the C^cQ setting. It is relatively straightforward to construct a "hybrid" KEM with quantum transitional security by performing a quantum-resistant KEM in addition to a classically secure KEM and deriving the shared key from both outputs. This approach preserves forward secrecy against future quantum adversaries while also maintaining classical security in the event that the security assumptions of the quantum-resistant KEM fail. The downside of schemes that are only secure in the C^cC or C^cQ settings is that they must be immediately replaced with post-quantum schemes after adversaries gain access to sufficiently powerful quantum computation.

In 2016, Google tested the deployment of a quantum transitionally secure key exchange in TLS [Lan16]. In a follow-up study in 2019, Google and Cloudflare jointly tested another quantum transitionally secure TLS ciphersuite using more recent quantum-resistant key exchanges [KV19]. They found that recent lattice-based schemes are so efficient that they effectively do not add any latency to most TLS handshakes.

DAKEZ, ZDH, and XZDH optionally provide quantum transitional security in the C^cQ setting by incorporating an Indistinguishability under Chosen Plaintext Attack (IND-CPA) quantum-resistant KEM. Because there are many different quantum-resistant KEMs with varying security properties, performance, and usage restrictions, the new DAKEs provide generic "placeholders" for a quantum-resistant KEM. An implementer is free to incorporate the KEM of their choice when using the new protocols. Section 5.7.1 returns to the practical issues surrounding this choice.

5.3 Efficient Cryptographic Primitive Constructions

The primary barrier to adoption of strongly deniable DAKEs like Φ_{dre}, RSDAKE, and Spawn is poor performance caused by the use of inefficient Dual-Receiver Encryption (DRE) and *ring signatures*

in the standard model. DRE [DLKY04] is similar to ordinary public-key encryption, except that messages are encrypted for two recipient public keys. The message can be decrypted by *either* corresponding private key, and it is verifiable that decrypting with either key produces the same result. DRE is similar to the more well-known notion of broadcast encryption [FN93], except that DRE does not require centralized generation of private keys, and broadcast encryption does not provide any verifiability guarantees. Ring signatures [RST01] are similar to ordinary digital signatures, except that messages are signed by a set of potential signers called a *ring*. Anyone with knowledge of a private key corresponding to *any* public key in this ring can produce the ring signature, and it is not possible to determine which key was used.

These two primitives are very useful in deniable key exchanges; ring signatures provide deniable authentication, and DRE can assist with online transcript forgery. Ring signatures with a carefully selected ring can convince a verifier of the authenticity of a message (due to a non-transferable belief that the signer cannot know certain keys) while allowing the signer to plausibly deny authorship (because the signature could have been produced using other keys in the ring). DRE can be used to allow a simulator to recover the shared secret and use it to forge subsequent messages even when outsourcing the generation of ephemeral secrets to an adversary; this technique enables online deniability by eliminating a potential protocol for producing evidence of an exchange.

The best known constructions of DRE and ring signatures with the appropriate security properties in the standard model are inefficient and make use of primitives such as cryptographic pairings. Pairing-based DRE constructions rely on stronger security assumptions than their ROM-based counterparts, and these assumptions are still occasionally the subject of newly discovered attacks [KB16]. This section specifies the DRE and ring signature primitives, the security properties that they must provide to construct the new DAKEs, and efficient ROM-based constructions of the primitives.

5.3.1 Notation

All of the definitions in this chapter are implicitly given with respect to a security parameter λ. When defining concrete two-party protocols, the initiator and the responder are denoted as $\mathcal{I}$ and $\mathcal{R}$, respectively. When written with quotes, "$\mathcal{P}$" denotes an implementation-defined identifier for the party $\mathcal{P}$.

5.3.2 Dual-Receiver Encryption with Associated Data

While it is possible to construct the new DAKEs using traditional DRE [DLKY04], transmission sizes can be reduced by incorporating Rogaway's notion of associated authenticated data [Rog02] in the primitive. The combined primitive is a new cryptosystem called Dual-Receiver Encryption with Associated Data (DREAD). A DREAD scheme consists of the following functions:

- **DRGen**(s): a key generation function. DRGen produces a key pair (pk, sk) for use with the scheme. s represents the random coins used to generate the key pair. Omitting s implies that s is set to newly sampled randomness.
- **DREnc**$(pk_1, pk_2, m, \Phi; r)$: an encryption function. DREnc encrypts a message m with associated data Φ under two public keys pk_1 and pk_2 using the random coins represented by r. If pk_1 and pk_2 are valid public keys, then DREnc produces a ciphertext γ. Otherwise, DREnc returns the special value $\perp$. The output of DREnc is consistent across invocations with the same $(pk_1, pk_2, m, \Phi; r)$ as input and varies when r is changed. Any value of r produces a valid encryption of m. Omitting r implies that r is set to newly sampled randomness.
- **DRDec**$(i, pk_1, pk_2, sk, \Phi, \gamma)$: a decryption function. For $i \in \{1, 2\}$, if (pk_i, sk) was generated by DRGen and $\gamma \neq \perp$ is a ciphertext $\gamma = \text{DREnc}(pk_1, pk_2, m, \Phi; r)$ for some r and m, then $\text{DRDec}(i, pk_1, pk_2, sk, \Phi, \gamma)$ returns m. In all other cases, DRDec returns $\perp$ with overwhelming probability.

The associated data Φ is information that is not transmitted with the ciphertext, but is nonetheless authenticated; the same value must be provided to both DREnc and DRDec in order for the decryption of m to succeed.

Note that the definitions of DREnc and DRDec implicitly rely on the ability to verify that a public key is valid and corresponds to a unique secret key. A scheme satisfying this property is called *admissible* [CFZ14]. This chapter considers only admissible schemes. Additionally, the new key exchange protocols require DREAD instantiations to satisfy two security properties that are naturally extended from the DRE security definitions introduced by Chow et al. [CFZ14] by incorporating the Φ parameter where necessary: the chosen DREAD scheme must be *sound* (decryption always produces the same value for both secret keys) and Indistinguishability under adaptive Chosen Ciphertext Attack (IND-CCA2). Section 5.3.2.1 details the necessary modifications to the definitions from Chow et al. [CFZ14], which merely involve allowing the adversary to select Φ in the security games.

The few explicit constructions of DRE in the literature are almost all designed for the standard model. Since random oracles are permitted in the new DAKE protocols, an extremely efficient

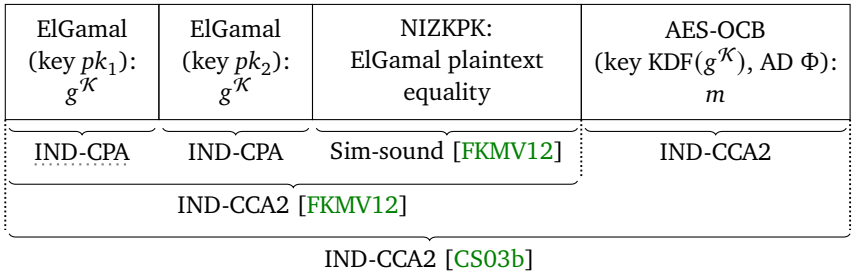


Figure 5.1 IND-CCA2 PROOF METHOD FOR DREAD CONSTRUCTION. The new construction is a hybrid encryption of message m for keys pk_1 and pk_2 with associated data Φ. (Ref: 86)

DREAD scheme can be constructed using the Naor-Yung paradigm [NY90] within a hybrid cryptosystem. Figure 5.1 depicts the new DREAD cryptosystem built using this approach. The ciphertext consists of a secret group element encrypted twice with ElGamal [ElG85] (once to each recipient), a Non-Interactive Zero-Knowledge Proof of Knowledge (NIZKPK) proving that the ElGamal ciphertexts contain the same plaintext, and an Authenticated Encryption with Associated Data (AEAD) of the message m using a symmetric key derived from the secret group element. This approach is known to generate a sound DRE system [CFZ14]. ElGamal was chosen because it is a very efficient scheme with the desired security properties, and it can be instantiated without adding new security assumptions to the new DAKEs. The NIZKPK is formed using the Fiat-Shamir transform [FS87]. When used in the Naor-Yung paradigm, Faust et al. [FKMV12] proved that such NIZKPKs are simulation-sound and thus yield IND-CCA2 security when combined with IND-CPA encryption schemes like ElGamal. Since an AEAD must be IND-CCA2 secure, the overall combination of schemes is also IND-CCA2 secure using the proof from Cramer and Shoup [CS03b, Th. 7.2].[6] In practice, the AEAD may be instantiated using AES-256 in OCB or an equivalent unpatented mode [HKM15] using the NIZKPK to derive the nonce. Section 5.3.2.1 shows that the soundness of the NIZKPK proof implies soundness of the DREAD construction.

The complete DREAD construction is as follows:

- **Setup**: all users share a group description, $(\mathbb{G}, q, g)$, for use in the ElGamal scheme [ElG85]. g is a generator for the group $\mathbb{G}$ of prime order q. All hash functions used in this chapter output values in $\mathbb{Z}_q$.
- **DRGen**(s): keys are generated as in the ElGamal scheme [ElG85]. If s is omitted, it is set to $s \xleftarrow{\$} \mathbb{Z}_q$. The resulting public key for a user is $h = g^s$, and the secret key is s.

[6] When expressing the ElGamal encryptions of $\mathcal{K}$ and the NIZKPK as the key encapsulation mechanism, AES-OCB as the one-time symmetric key encryption scheme, and the IND-CCA2 security game as in Section 5.3.2.1, the proof directly applies.

- **DREnc**$(pk_1, pk_2, m, \Phi; r)$: If r is omitted, then it is set to $r \xleftarrow{\$} \mathbb{Z}_q$[5]. In any case, r is interpreted as $r = (\mathcal{K}, k_1, k_2, t_1, t_2)$. k_1 and k_2 are used to encrypt $g^{\mathcal{K}}$ for each recipient using ElGamal [ElG85], and t_1 and t_2 are used later in the NIZKPK.[7] If pk_1 or pk_2 are outside of $\mathbb{G}$ or are the identity element, DREnc returns $\perp$. Otherwise, the resulting ciphertexts consist of $c_{1i} = g^{k_i}$ and $c_{2i} = pk_i^{k_i} \cdot g^{\mathcal{K}}$ for $i \in \{1, 2\}$. The message m is encrypted using the AEAD scheme with key KDF$(g^{\mathcal{K}})$, where KDF is secure key derivation function, and associated data Φ, denoted by $\Theta = \text{AEnc}(\text{KDF}(g^{\mathcal{K}}), m, \Phi)$. The result also includes a NIZKPK of the following statement, proving that the ciphertext is well formed, given in Camenisch-Stadler notation [CS97]:

$$PK\left\{(k_1, k_2) : c_{11} = g^{k_1} \wedge c_{12} = g^{k_2} \wedge \frac{c_{21}}{c_{22}} = \frac{pk_1^{k_1}}{pk_2^{k_2}}\right\}$$

  The party calling DREnc acts as the prover $\mathcal{P}$ for the NIZKPK. $\mathcal{P}$ uses the random values $t_i \in \mathbb{Z}_q$ for $i \in \{1, 2\}$ to compute $T_{11} = g^{t_1}$, $T_{12} = g^{t_2}$, and $T_2 = {pk_1^{t_1}}/{pk_2^{t_2}}$. Next, $\mathcal{P}$ computes the hash:

$$L = H(g \| q \| pk_1 \| pk_2 \| c_{11} \| c_{21} \| c_{12} \| c_{22} \| T_{11} \| T_{12} \| T_2 \| \Phi)$$

  where H is a cryptographic hash function modeled by a random oracle.[8] $\mathcal{P}$ then computes $n_i = t_i - L \cdot k_i \pmod q$ for $i \in \{1, 2\}$. DREnc returns the encryption of m, $\gamma = (c_{11}, c_{21}, c_{12}, c_{22}, L, n_1, n_2, \Theta)$.

- **DRDec**$(i, pk_1, pk_2, sk_i, \Phi, \gamma)$: the recipient parses γ to retrieve its components. If either public key is outside of $\mathbb{G}$ or is the identity element, or if γ is not of the correct form, then DRDec returns $\perp$. Otherwise, the recipient computes the following three values:

$$T'_{11} = g^{n_1}(c_{11})^L \qquad T'_{12} = g^{n_2}(c_{12})^L \qquad T'_2 = \frac{pk_1^{n_1}}{pk_2^{n_2}}\left(\frac{c_{21}}{c_{22}}\right)^L$$

[7] The IND-CPA security of ElGamal holds only when the "plaintext" is a group element. For this reason, directly encrypting a random scalar such as $\mathcal{K}$ would be insecure in most circumstances without additional security assumptions. This topic is revisited in Section 8.2.4.2, which presents a scheme that encrypts "raw" scalars and discusses various techniques to make this approach secure.

[8] If H is instantiated with a concrete hash function that suffers from length-extension or padding attacks [Tsu92] (e.g., SHA-2), then Φ should be hashed before it is passed as input to H, thereby ensuring that the input is of fixed length. This additional hash is unnecessary if Φ is unused, is already of fixed length, or when using an appropriate hash function (e.g., SHA-3).

The recipient then computes L' using the same hash operation described in DREnc:

$$L' = H(g \| q \| pk_1 \| pk_2 \| c_{11} \| c_{21} \| c_{12} \| c_{22} \| T'_{11} \| T'_{12} \| T'_2 \| \Phi)$$

If $L \neq L'$, DRDec returns $\perp$. Otherwise, the recipient recovers the secret group element $g^{\mathcal{K}} = {}^{c_{2i}}/_{c_{1i}^{sk_i}}$. The recipient can then recover the message m by decrypting the AEAD ciphertext Θ with key $\text{KDF}(g^{\mathcal{K}})$ and associated data Φ, denoted by $m = \text{ADec}(\text{KDF}(g^{\mathcal{K}}), \Theta, \Phi)$. If ADec fails, DRDec returns $\perp$. Otherwise, it returns m.

5.3.2.1 Proof of Security

Security properties for DREAD schemes can be easily derived from those for DRE schemes given by Chow et al. [CFZ14] by incorporating the associated data parameter Φ where needed:

- **Soundness**[9]: ciphertexts decrypt to the same value (including $\perp$) even when the keys used by DREnc are not honestly generated, or the ciphertext is not produced by DREnc at all. Concretely, any Probabilistic Polynomial Time (PPT) adversary has negligible advantage in the following game:

  1. The adversary produces a ciphertext γ, associated data Φ, and two public keys pk_1 and pk_2.
  2. Let sk_1 and sk_2 be the unique secret keys associated with pk_1 and pk_2.
  3. The adversary wins if $\text{DRDec}(1, pk_1, pk_2, sk_1, \Phi, \gamma) \neq \text{DRDec}(2, pk_1, pk_2, sk_2, \Phi, \gamma)$.

  Note that this definition requires the cryptosystem used in the DREAD construction to be *admissible*, as defined in Section 5.3.2. ElGamal is admissible [CFZ14].

- **Dual-receiver IND-CCA2 security**: any PPT adversary has negligible advantage in the following game against a sound scheme[10]:

  1. The challenger produces $(pk_i, sk_i) \leftarrow \text{DRGen}()$ for $i \in \{1, 2\}$ and sends (pk_1, pk_2) to the adversary.
  2. The adversary is given decryption oracle access for $\text{DRDec}(1, pk_1, pk_2, sk_1, \cdot, \cdot)$. The adversary may perform a polynomially bounded number of DREnc calls, oracle requests, and other operations.

[9] Chow et al. [CFZ14] refer to the analogous property for DRE as *strong soundness*.

[10] The soundness property simplifies the definition of the game by eliminating the need for two decryption oracles [CFZ14].

3. The adversary chooses two messages, m_1 and m_2, of equal length, and associated data Φ. The adversary sends (m_1, m_2, Φ) to the challenger.
4. The challenger chooses $b \xleftarrow{\$} \{1, 2\}$ and sends $\gamma \leftarrow \text{DREnc}(pk_1, pk_2, m_b, \Phi)$ to the adversary.
5. The adversary may perform a polynomially bounded number of calls to DREnc, oracle requests, and other operations. The adversary immediately loses if it ever queries the oracle for $\text{DRDec}(1, pk_1, pk_2, sk_1, \Phi, \gamma)$.
6. The adversary outputs a guess b'.
7. The adversary wins if $b = b'$.

It is easy to see that the DREAD construction in Section 5.3.2 satisfies the soundness property. If the NIZKPK verifies, then except with negligible probability, both ElGamal ciphertexts must encode the same value $g^{\mathcal{K}}$ due to the soundness of the NIZKPK scheme. Consequently, both sk_1 and sk_2 must cause ADec to return the same value. Otherwise, decryption with either key returns $\perp$.

The proof of dual-receiver IND-CCA2 security for the new DREAD construction can be derived from the proofs given by Faust et al. [FKMV12] according to the method described in Section 5.3.2.

5.3.3 Efficient Ring Signatures for Three Keys

One way to achieve strong deniability in a DAKE is to use ring signatures, as in RSDAKE and Spawn [UG15]. Unfortunately, the literature lacks an explicit ring signature construction that provably provides the security properties required by the new DAKEs while also being efficient, even though the techniques to do so are known. Schemes introduced prior to the publication of the security definitions from Bender et al. [BKM06] lack the appropriate proofs [AOS02], require unusual primitives that are difficult to use in practice [BGLS03; DKNS04; ZK02], or reduce to a generic form of the approach described in this section [AHR05]. Schemes published after the definitions from Bender et al. [BKM06] either focus on adding new features, or offering strong standard model security proofs (at the expense of practicality) [XQL13]. Additionally, "efficient" ring signature schemes typically focus on scalability with respect to ring size [DKNS04]. The new DAKEs described in this chapter use small rings with only three potential signers, so scalable schemes are usually not optimized for this setting. There are also a variety of well-known ring signature schemes that operate in slightly different scenarios, at the cost of performance [BSS02; LWW04; WMZW11; CYH05].

RSDAKE and Spawn use ring signatures to prove that the signer knows one of three private keys. It is possible to construct an efficient ROM-based ring signature scheme to accomplish this by issuing a Signature of Knowledge (SoK) of one out of three discrete logarithms [CS97]. An SoK is a non-interactive zero-knowledge proof system demonstrating knowledge of a value. The ring signature presented in this section is constructed using an SoK based on the Schnorr signature scheme [Sch91] and the "OR proof" technique introduced by Cramer et al. [CDS94].[11] The Fiat-Shamir transform [FS87] makes this proof non-interactive. This approach is far more efficient than the ring signature scheme used in RSDAKE and Spawn [SW07].

Assume that each public key is of the form $A_i = g^{a_i}$, as in a typical DH key exchange (where g is the generator for a group $\mathbb{G}$ of prime order q). The SoK over keys (A_1, A_2, A_3) is a proof of the following statement, given in Camenisch-Stadler notation [CS97]:

$$SKREP\{(a) : g^a = A_1 \vee g^a = A_2 \vee g^a = A_3\}(m)$$

Assuming without loss of generality that the signer knows a_1, the proof proceeds as follows:

1. Generate random values $t_1, c_2, c_3, r_2, r_3 \in \mathbb{Z}_q$.
2. Compute $T_1 = g^{t_1}$.
3. Compute $T_2 = g^{r_2} A_2^{c_2}$ and $T_3 = g^{r_3} A_3^{c_3}$.
4. Compute $c = H(g \| q \| A_1 \| A_2 \| A_3 \| T_1 \| T_2 \| T_3 \| m)$, where H is a hash function modeled by a random oracle and m is the message to "sign".
5. Compute $c_1 = c - c_2 - c_3 \pmod q$.
6. Compute $r_1 = t_1 - c_1 a_1 \pmod q$.

The resulting proof consists of $(c_1, r_1, c_2, r_2, c_3, r_3)$. To verify the proof, the verifier begins by computing $c' = H(g \| q \| A_1 \| A_2 \| A_3 \| g^{r_1} A_1^{c_1} \| g^{r_2} A_2^{c_2} \| g^{r_3} A_3^{c_3} \| m)$. The verifier then checks whether $c' \stackrel{?}{=} c_1 + c_2 + c_3 \pmod q$.

In the general case where the prover knows a secret a_i, they select t_i randomly, compute $T_i = g^{t_i}$, and compute $T_j = g^{r_j} A_j^{c_j}$ for $j \neq i$, proceeding as normal and ultimately computing c_i and r_i. The order of elements passed to H and sent to the verifier must *not* depend on the secret known to the prover (otherwise, the key used to produce the proof can be inferred in practice).

[11] This technique was subsequently extended [CPS+16b; CPS+16a]; however, these results are not useful in the ring signature construction because the statement to prove is fully known when the proof commences.

The SoK can be used to construct a ring signature scheme that will authenticate participants in the new DAKE protocols. The ring signature scheme consists of the following functions:

- **RSig**$(A, a, S, m; r)$: RSig produces an SoK σ, bound to the message m, that demonstrates knowledge of a private key corresponding to one of three public keys. S is the ring of public keys $\{A_1, A_2, A_3\}$ that could possibly have produced the proof. It is required that (A, a) is a DH keypair in the appropriate group, and $A \in S$. r contains the random coins used to compute the output. The SoK is computed as described above, with r interpreted as $r = (t_i, c_j, c_k, r_j, r_k)$, where i is the index of A in S, and j and k are the values in $\{1, 2, 3\}$ such that i, j, and k are distinct and $j < k$. If r is omitted, then r is set to $r \xleftarrow{\$} \mathbb{Z}_q$[5].
- **RVrf**(S, σ, m): a verification function. RVrf returns TRUE if the SoK σ is valid, and FALSE otherwise. Correctness requires that RVrf(S, σ, m) = TRUE when σ = RSig$(A, a, S, m; r)$ for any valid inputs.

Bender et al. [BKM06] defined several security properties for ring signature schemes. The new DAKEs require RSig and RVrf to exhibit *anonymity against full key exposure* (the signer remains anonymous within the ring even if all signing keys are subsequently compromised) and *unforgeability with respect to insider corruption* (signatures are unforgeable even if signing keys outside of the ring are adversarially generated). The next section includes the definitions of these properties and the security proofs for the RSig/RVrf ring signature scheme.

5.3.3.1 Proof of Security

The RSig/RVrf scheme presented in Section 5.3.3 must satisfy the following two security properties defined by Bender et al. [BKM06]:

- **Anonymity against full key exposure** [BKM06, Def. 4]: it is not possible to determine which secret key was used to produce the ring signature, even if all secret keys are revealed. Concretely, any PPT adversary has negligible advantage in the following game:
  1. The challenger generates n key pairs (PK_i, SK_i) where n is a polynomial of the security parameter. Let PK be the set of all public keys, and SK be the set of all secret keys.
  2. The adversary is given PK and access to an oracle $O_{\text{RSig}}(\cdot, \cdot, \cdot)$ such that $O_{\text{RSig}}(\bar{S}, \bar{m}, \bar{i})$ returns RSig$(PK_{\bar{i}}, SK_{\bar{i}}, \bar{S}, \bar{m})$ such that $PK_{\bar{i}} \in \bar{S}$.
  3. The adversary outputs a message m, distinct indices i and j, and a ring S for which $PK_i, PK_j \in S$. The adversary is given SK.

4. The challenger chooses $b \xleftarrow{\$} \{i, j\}$ and sends $\sigma \leftarrow \mathrm{RSig}(PK_b, SK_b, S, m)$ to the adversary.
5. The adversary outputs b' and wins if $b = b'$.

- **Unforgeability with respect to insider corruption** [BKM06, Def. 7]: it is not possible to produce an illegitimate ring signature, even with access to legitimate signatures that were produced using adversarially controlled keys in their rings. Concretely, any PPT adversary wins the following game with negligible probability:

1. The challenger generates n key pairs (PK_i, SK_i) where n is a polynomial of the security parameter. Let PK be the set of all public keys, SK be the set of all secret keys, and $C \leftarrow \emptyset$ be the set of corrupted users.
2. The adversary is given PK.
3. The adversary is given access to an oracle $O_{\mathrm{RSig}}(\cdot, \cdot, \cdot)$ such that $O_{\mathrm{RSig}}(\bar{S}, \bar{m}, \bar{i})$ returns $\mathrm{RSig}(PK_{\bar{i}}, SK_{\bar{i}}, \bar{S}, \bar{m})$ such that $PK_{\bar{i}} \in \bar{S}$.
4. The adversary is given access to a corruption oracle $O_{\mathrm{corr}}(\cdot)$ such that $O_{\mathrm{corr}}(\bar{i})$ returns $SK_{\bar{i}}$ and sets $C \leftarrow C \cup \{PK_{\bar{i}}\}$.
5. The adversary outputs (S, σ, m) and wins if $\mathrm{RVrf}(S, \sigma, m) = \mathrm{TRUE}$, the adversary never queried $O_{\mathrm{RSig}}(S, m, \cdot)$, and $S \subseteq PK \setminus C$.

Some previous work has been done on proving the security of ring signatures in the universal composability framework. Fischlin and Onete [FO11] presented ideal functionalities for SoKs, and Yoneyama and Ohta [YO07] defined a ring signature functionality that is provably equivalent to Bender's security properties [BKM06]. However, neither of these schemes is defined for recent frameworks with more accurate modeling of cross-session state (e.g., long-lived public keys). Defining ring signature functionalities for more realistic frameworks is a substantial task that is beyond the scope of this work. The remainder of this section presents security theorems for the RSig/RVrf scheme following the game-based definitions.

Theorem 1 | Anonymity of RSig/RVrf

If the SoK produced by RSig is zero-knowledge, then the RSig/RVrf ring signature scheme provides anonymity against full key exposure. (Ref: 92)

The proof of Theorem 1 directly follows from the security assumption. Since the SoK is zero-knowledge, the adversary learns nothing from the signing oracle or from the private keys. The construction of the SoK in Section 5.3.3 is known to be zero-knowledge in the ROM [CS97]. In fact, the RSig/RVrf construction fulfills an even stronger property than required: the signer remains anonymous even if the secret keys are revealed at the start of the game.

Theorem 2 | Unforgeability of RSig/RVrf

If the discrete logarithm problem is hard in $\mathbb{G}$, then the RSig/RVrf ring signature scheme provides unforgeability with respect to insider corruption. (Ref: 93)

It is known that SoKs produced by the Fiat-Shamir heuristic [FS87] are *weakly simulation-extractable* [FKMV12]: for any PPT adversary with access to a proof simulator that can produce a valid new proof (one not retrieved from the simulator), there is an efficient extractor that can retrieve a witness for that proof. Using this extractor, the reduction for Theorem 2 becomes simple: given an adversary that can forge ring signatures, include a blinded group element in the ring, then extract a witness from the forged proof to compute the discrete logarithm of that group element with non-negligible probability.

5.3.4 Quantum-Resistant Key Encapsulation

As described in Section 5.2.3, the new DAKE protocols optionally provide quantum transitional security against future quantum adversaries. This is accomplished by using a black-box passively secure quantum-resistant KEM modeled by the following functions:

- **QRGen**(s): a key generation function for initiators. QRGen produces a key pair (PQ_I, SQ_I) for use with the scheme. s represents the random coins used to generate the key pair and may be omitted to denote that s is set to newly sampled randomness.
- **QREncaps**$(PQ_I; s)$: an encapsulation function used by responders. QREncaps takes an initiator public key as input and produces a response message Q_R and a shared secret Q_k. s represents the random coins used to generate the encapsulation and may be omitted to denote that s is set to newly sampled randomness.
- **QRDecaps**(SQ_I, Q_R): a decapsulation function used by initiators to derive the shared key. QRDecaps takes a private key SQ_I and a response message Q_R as input and returns either a shared secret Q_k or the special symbol $\perp$ to indicate a failure condition.

This model permits a variety of quantum-resistant KEMs to be used in the new DAKEs.[12] For contributory schemes, QREncaps uses PQ_I to derive Q_k, and Q_R is a public contribution from the

[12] The candidate quantum-resistant KEMs in the National Institute of Standards and Technology (NIST) [AAA+20] standardization effort are all expressed in terms of these three functions. However, the names of the functions and variables differ slightly from the NIST notation because the DAKEs in this chapter were developed prior to the standardization of the notation.

responder. For non-contributory (key transport) schemes, the responder unilaterally determines Q_k, and Q_R securely delivers Q_k to the initiator. If $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$ and $(Q_R, Q'_k) \leftarrow \text{QREncaps}(PQ_I)$, then a *correct* scheme will ensure that $\text{QRDecaps}(SQ_I, Q_R)$ produces $Q_k = Q'_k$. Some schemes do not guarantee correctness: they allow QRDecaps to return $\perp$ for valid inputs, but with only negligible probability. A scheme is said to be δ-*correct* [HHK17, §2.2] if:

$$\Pr\left[\text{QRDecaps}(SQ_I, Q_R) \neq Q_k \;\middle|\; (PQ_I, SQ_I) \leftarrow \text{QRGen}(); (Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)\right] \leq \delta$$

The new DAKE protocols perform both a black-box quantum-resistant KEM and a traditional DH key exchange, then input both shared secrets to a key derivation function. A similar combined KEM technique is used by CECPQ1 [Lan16] and CECPQ2 [KV19]. For a given security parameter λ, the quantum-resistant KEM must be $2^{-\lambda}$-correct and IND-CPA secure. Since the probability of QRDecaps failing for a valid encapsulation is negligible, it can be ignored in the protocol design. The security propositions and theorems in this chapter restrict their attention to classical security. The quantum transitional security properties of the schemes are discussed separately in Section 6.2.4.1.

To disable the optional quantum transitional security feature, the functions can be instantiated with "no-op" versions that produce values with zero length, denoted by $\varnothing$:

- **QRGen**$^{\varnothing}(s)$: returns $PQ_I = \varnothing$ and $SQ_I = \varnothing$.
- **QREncaps**$^{\varnothing}(PQ_I; s)$: returns $Q_R = \varnothing$ and $Q_k = \varnothing$.
- **QRDecaps**$^{\varnothing}(SQ_I, Q_R)$: returns $Q_k = \varnothing$.

In terms of abstract protocols, the interpretation of these values is that transmission of $PQ_I = \varnothing$ and $Q_R = \varnothing$ is free, and concatenating $Q_k = \varnothing$ with another value does not alter that value. In practice, implementations that do not use quantum transitional security can simply exclude these function calls from the DAKEs.

5.4 DAKEZ

Given the cryptographic primitives defined in Section 5.3, it is now possible to define the first new DAKE: DAKEZ. Figure 5.2 depicts the protocol. KDF refers to a secure key derivation function that is distinct from the KDF used in the DREAD construction.

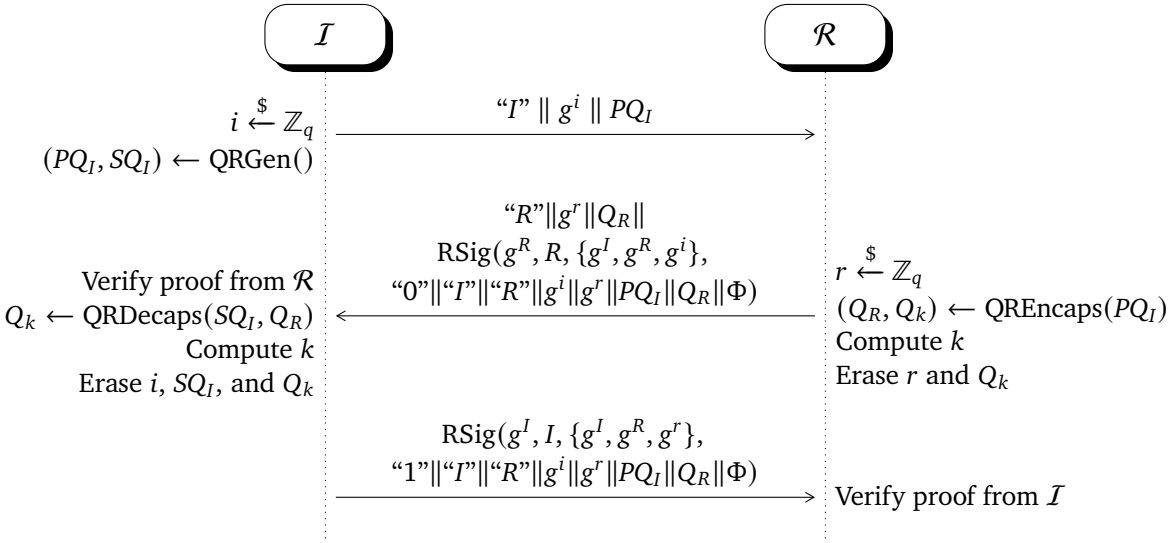


Figure 5.2 DAKEZ KEY EXCHANGE PROTOCOL. Φ is shared session state. The shared secret is $k = \text{KDF}(g^{ir} \| Q_k)$.
(Refs: 94 and 122)

Initially, the developer selects a common group $\mathbb{G}$ generated by g with prime order q. The CDH problem should be hard within $\mathbb{G}$. Initiator $\mathcal{I}$ chooses long-term secret key I and public key g^I. Responder $\mathcal{R}$ chooses long-term secret key R and public key g^R. All parties in the system are expected to have securely distributed their long-term public keys before the start of the protocol session.[13]

A DAKEZ session normally takes place within a higher-level protocol (e.g., XMPP or HTTP). To prevent attacks that rebind the DAKEZ transcript into different contexts, it is prudent to ensure that the DAKEZ session authenticates its context. Given state information Φ associated with the higher-level context, DAKEZ authenticates that both parties share the same value for Φ. Section 5.7 discusses the contents of this state information in practice. A DAKEZ session proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$. $\mathcal{I}$ sends "I", g^i, and PQ_I to $\mathcal{R}$.

[13] This assumption simplifies the protocol definitions and security proofs. In practice, the implementation-defined identifier "$\mathcal{P}$" for a party $\mathcal{P}$ can include the long-term public key, and these keys can be verified using a trust establishment mechanism (see Section 2.6) at some point before or after the DAKE session. Section 5.7 discusses how to integrate the DAKEs with secure messaging protocols in greater detail.

2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$. $\mathcal{R}$ sends "R", g^r, Q_R, and $\text{RSig}(g^R, R, \{g^I, g^R, g^i\}, t)$ to $\mathcal{I}$, where the tag t is $t = \text{"0"}\|\text{"}I\text{"}\|\text{"}R\text{"}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{R}$ computes $k = \text{KDF}((g^i)^r\|Q_k)$ and securely erases r and Q_k.

3. $\mathcal{I}$ verifies the proof sent by $\mathcal{R}$. $\mathcal{I}$ sends $\text{RSig}(g^I, I, \{g^I, g^R, g^r\}, t)$ to $\mathcal{R}$, where the tag t is $t = \text{"1"}\|\text{"}I\text{"}\|\text{"}R\text{"}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{I}$ computes $Q_k = \text{QRDecaps}(SQ_I, Q_R)$ and uses it to compute the shared secret $k = \text{KDF}((g^r)^i\|Q_k)$, then securely erases i, SQ_I, and Q_k.

4. $\mathcal{R}$ verifies the proof sent by $\mathcal{I}$.

This algebraic description of DAKEZ omits several important practical considerations that must be handled correctly to produce a secure implementation. Section 5.7 discusses these considerations.

Proposition 1 states a simplified security theorem for DAKEZ. A formal theorem and proof sketch appear in Section 6.2.

Proposition 1 | DAKEZ is secure (informal)

If the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then DAKEZ provides the security properties listed in Section 5.2.2 against adaptive corruptions. (Refs: 96 and 502)

5.5 ZDH

Spawn [UG15] is a two-flow DAKE with a single post-specified peer [CK02a] (i.e., the initiator $\mathcal{I}$ does not know the identity of its partner a priori, but the responder $\mathcal{R}$ does). In an interactive setting, Spawn provides online deniability for both parties. In non-interactive settings where the adversary is able to guarantee that the first message is honestly generated by the true $\mathcal{I}$, Spawn does not provide online deniability for $\mathcal{R}$ [UG15]. In these settings, *explicitly* sacrificing online deniability for $\mathcal{R}$ results in a more efficient protocol, ZDH.

5.5.1 Efficient Spawn with Random Oracles

Before introducing the construction of ZDH, it is insightful to first instantiate Spawn [UG15] using the DREAD and ring signature schemes presented in Section 5.3. While improving the performance of Spawn is not a primary goal of this work, doing so is important for three reasons:

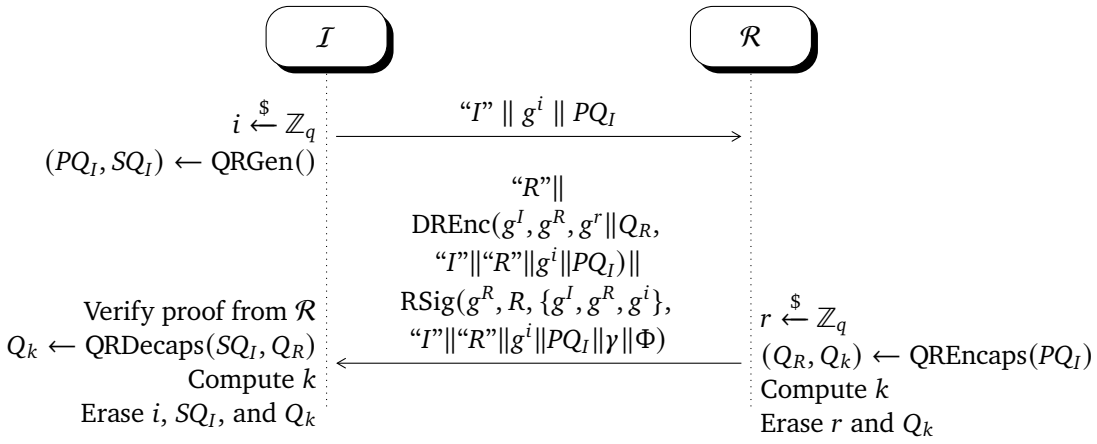


Figure 5.3 SPAWN$^+$ KEY EXCHANGE PROTOCOL. Φ is shared session state. γ is the DREnc output. The shared secret is $k = \text{KDF}(g^{ir} \| Q_k)$. (Ref: 97)

it facilitates a precise characterization of the design improvements in ZDH, it simplifies the security proof of ZDH, and it provides a baseline performance comparison for ZDH. Figure 5.3 depicts Spawn$^+$, a contributory instantiation of Spawn using the ROM-based primitives. Unlike DAKEZ, which is secure in the same model as RSDAKE, the contributiveness of Spawn$^+$ makes it provably secure against stronger adversaries than Spawn.

Initially, the developer selects a common group $\mathbb{G}$ generated by g with prime order q in which the CDH problem is hard. Initiator $\mathcal{I}$ and responder $\mathcal{R}$ generate long-term key pairs (g^I, I) and (g^R, R), respectively, as described in Section 5.3. All parties in the system are expected to have securely distributed their long-term public keys before the start of the protocol session. A Spawn$^+$ session between $\mathcal{I}$ and $\mathcal{R}$ within a higher-level protocol with shared session state Φ proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$. $\mathcal{I}$ sends $(\text{"}I\text{"}, g^i, PQ_I)$ to $\mathcal{R}$. An untrusted server may cache this "prekey" message.

2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$. $\mathcal{R}$ then computes the ciphertext $\gamma = \text{DREnc}(g^I, g^R, g^r \| Q_R, t)$, where the tag t is given by $t = \text{"}I\text{"}\|\text{"}R\text{"}\|g^i\|PQ_I$. $\mathcal{R}$ computes $\sigma = \text{RSig}(g^R, R, \{g^I, g^R, g^i\}, t\|\gamma\|\Phi)$. $\mathcal{R}$ sends $(\text{"}R\text{"}, \gamma, \sigma)$ to $\mathcal{I}$. $\mathcal{R}$ computes $k = \text{KDF}((g^i)^r \| Q_k)$ and securely erases r and Q_k. Note that $\mathcal{R}$ can attach an initial message m to this flow by immediately encrypting it with a symmetric cryptosystem keyed with k.

3. $\mathcal{I}$ verifies the proof σ and decrypts γ using I. $\mathcal{I}$ verifies that the decrypted message is of the correct form (e.g., the fields are of the expected length) and that the prekey (g^i, PQ_I) that $\mathcal{I}$

previously sent remains unused. $\mathcal{I}$ computes $Q_k = \text{QRDecaps}(SQ_I, Q_R)$ and the shared secret $k = \text{KDF}((g^r)^i \| Q_k)$, then securely erases i, SQ_I, and Q_k. If an initial message was attached, $\mathcal{I}$ can decrypt the message using k.

Developers should incorporate the safeguards discussed in Section 5.7 to securely implement Spawn$^+$.

Proposition 2 states a simplified security theorem for Spawn$^+$. A formal theorem and proof sketch appear in Chapter 6.

Proposition 2 | Spawn$^+$ is secure (informal)

If the DREAD scheme is sound and IND-CCA2 secure, the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then Spawn$^+$ provides the security properties listed in Section 5.2.2 against adaptive corruptions. (Ref: 98)

5.5.2 The ZDH Protocol

This section discusses the construction of ZDH, a DAKE that outperforms the ROM-based variant of Spawn$^+$ in non-interactive settings primarily by avoiding the use of DREAD operations to facilitate online deniability for $\mathcal{R}$. ZDH is depicted in the unshaded portions of Figure 5.4 (the shaded values are used only in XZDH, which is discussed in Section 5.6).

The initial ZDH setup is the same as for Spawn$^+$; parties generate and distribute long-term ElGamal public keys in a group $\mathbb{G}$ generated by g with prime order q in which the CDH problem is hard. Within a higher-level protocol with shared session state Φ, a ZDH session between initiator $\mathcal{I}$ and responder $\mathcal{R}$ with respective key pairs (g^I, I) and (g^R, R) proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$. $\mathcal{I}$ sends $(\text{“}I\text{”}, g^i, PQ_I)$ to $\mathcal{R}$. An untrusted server may cache this “prekey” message.

2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$. $\mathcal{R}$ then derives the shared keys $\kappa = \text{KDF}_1((g^i)^r \| (g^I)^r \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and $k = \text{KDF}_3(\kappa)$, where KDF_1, KDF_2, and KDF_3 are key derivation functions modeled by a random oracle. $\mathcal{R}$ securely erases r and Q_k. Let the tag t be defined by $t = \text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{R}$ computes $mac = \text{MAC}(M_k, t)$, where $\text{MAC}(key, msg)$ refers to a key-only unforgeable [FGSW16] message authentication code function for message msg with key key. $\mathcal{R}$ computes $\sigma =$

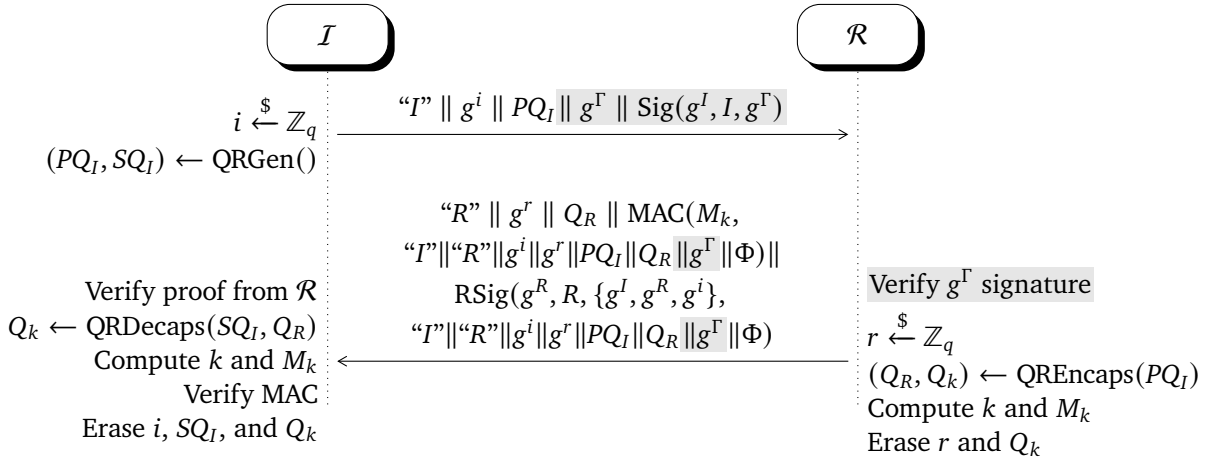


Figure 5.4 ZDH AND XZDH KEY EXCHANGE PROTOCOLS. Φ is shared session state. $\kappa = \text{KDF}_1(g^{ir}\|g^{\Gamma r}\|g^{Ir}\|Q_k)$, $M_k = \text{KDF}_2(\kappa)$ and the shared secret is $k = \text{KDF}_3(\kappa)$. Shaded terms are used in XZDH only, and omitted for ZDH. In XZDH, g^Γ is a reusable signed prekey. (Refs: 98 and 101 [a] [b])

$\text{RSig}(g^R, R, \{g^I, g^R, g^i\}, t)$. $\mathcal{R}$ sends ("R", g^r, Q_R, mac, σ) to $\mathcal{I}$. Note that $\mathcal{R}$ can attach a message m to this flow by immediately encrypting it with a symmetric cryptosystem keyed with k.

3. $\mathcal{I}$ verifies σ. $\mathcal{I}$ verifies that (g^i, PQ_I) is an unused prekey previously sent by $\mathcal{I}$. $\mathcal{I}$ computes $Q_k = \text{QRDecaps}(SQ_I, Q_R)$, $\kappa = \text{KDF}_1((g^r)^i \,\|\, (g^r)^I \,\|\, Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and $k = \text{KDF}_3(\kappa)$. $\mathcal{I}$ verifies the MAC and securely erases i, SQ_I, and Q_k. If a message was attached, $\mathcal{I}$ can decrypt it using k.

Developers should incorporate the safeguards discussed in Section 5.7 to securely implement ZDH.

Proposition 3 states a simplified security theorem for ZDH. A formal theorem and proof sketch appear in Chapter 6.

Proposition 3 | ZDH is secure (informal)

If the MAC is weakly unforgeable under chosen message attack [BKR00], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then ZDH provides the security properties listed in Section 5.2.2 against adaptive corruptions when the response is sent by an honest party. (Refs: 99 and 100)

5.5.3 Design Discussion

The primary difference between ZDH and Spawn$^+$ is that ZDH does not use DREAD to enable $\mathcal{R}$ to derive the shared secret during online deniability attacks. ZDH also uses a MAC to achieve full key confirmation for $\mathcal{I}$ [FGSW16]. Without this MAC, ZDH would be vulnerable to an identity misbinding attack [Kra03]; an active adversary $\mathcal{P}$ that replaced the ring signature with one for the ring $\{g^I, g^P, g^i\}$ would cause $\mathcal{I}$ to believe that it was communicating with $\mathcal{P}$ while sharing a key with $\mathcal{R}$.

In order to derive the shared secret for Spawn$^+$ from the exchanged information, a party must either know $\mathcal{R}$'s ephemeral secret, or $\mathcal{I}$'s ephemeral secret *and* either of the long-term secrets. In order to derive the shared secret for ZDH from the exchanged information, a party must know either $\mathcal{R}$'s ephemeral secret, or $\mathcal{I}$'s ephemeral and long-term secrets. The important difference is that in ZDH, a party that knows $\mathcal{R}$'s long-term secret and $\mathcal{I}$'s ephemeral secret cannot derive the key. This is precisely the knowledge available to $\mathcal{R}$ when it is trying to simulate $\mathcal{I}$ for an interactive third party. For this reason, the protocol explicitly lacks online deniability for $\mathcal{R}$, just like non-interactive Spawn and Spawn$^+$. This concession causes the prohibition of corrupt responders in Proposition 3, but also enables ZDH's efficient design.

ZDH's derivation of κ is similar to the derivation of the shared secret in 3DH [Mar13]. However, 3DH also includes g^{iR} as input to its KDF. The purpose of this term in 3DH is to authenticate $\mathcal{R}$, but ZDH accomplishes this using a ring signature instead. Moreover, including this term in ZDH would break online deniability for $\mathcal{I}$, as it does in 3DH, because an interactive third party that chooses i and coerces $\mathcal{I}$ into sending g^i can derive the shared secret while preventing $\mathcal{I}$ from doing so.

5.6 XZDH

Spawn, 3DH, Spawn$^+$, and ZDH all share a similar weakness: they are two-flow protocols with weak forward secrecy. This makes them vulnerable to an attack where an active adversary modifies the first flow from $\mathcal{I}$ to use an adversarially controlled ephemeral key, captures and drops the response from $\mathcal{R}$, and then compromises $\mathcal{I}$'s long-term secret key [BPR00]. $\mathcal{I}$ will never see the message, and the adversary will be able to decrypt it. Moreover, since long-term keys are usually meant to last for years, a long time may pass between $\mathcal{R}$ sending the message and the adversary compromising $\mathcal{I}$'s key. In practice, this attack requires a powerful adversary.

Signal's X3DH protocol [MP16] somewhat mitigates this weakness in 3DH by introducing the notion of *signed prekeys*. In contrast to the one-time prekeys used by 3DH, signed prekeys

are signed by long-term secret keys and are reusable. Each user maintains one signed prekey on the prekey server, which is changed on a roughly weekly basis [MP16]. The combination of a one-time prekey and a signed prekey is called a *prekey bundle*. X3DH incorporates a DH exchange between $\mathcal{I}$'s signed prekey and $\mathcal{R}$'s ephemeral key, and makes one-time prekeys optional. The benefit is that the aforementioned attack is thwarted if $\mathcal{I}$ changes its signed prekey and erases the old one before being corrupted by the adversary. Because the prekey is signed, it cannot be adversarially altered, and $\mathcal{I}$ controls the timing of key erasure. However, the use of signed prekeys in X3DH severely harms its offline deniability: transcripts can be forged only by one of the alleged participants, since the forger must complete a DH exchange between $\mathcal{I}$'s signed prekey and $\mathcal{R}$'s long-term key.

XZDH is a variant of ZDH incorporating signed prekeys. $\mathcal{I}$'s signed prekey is g^{Γ}, with corresponding secret key $\Gamma \in \mathbb{Z}_q$. $\mathcal{I}$ uploads g^{Γ} to the prekey server alongside an existentially unforgeable digital signature [GMR88] for g^{Γ} created using I as the signing key, denoted by $\text{Sig}(g^I, I, g^{\Gamma})$. The public verification function $\text{SVerif}(g^I, m, \xi)$ returns TRUE if ξ is a valid signature for message m created using I, and FALSE otherwise.

XZDH is identical to ZDH, except that it modifies κ to be $\kappa = \text{KDF}_1(g^{ir} \| g^{\Gamma r} \| g^{Ir} \| Q_k)$ and includes g^{Γ} in its MAC and RSig messages, as depicted in Figure 5.4. The first flow in Figure 5.4 represents the complete prekey bundle that $\mathcal{R}$ downloads from the server. g^{Γ} is reused across sessions, but $\mathcal{I}$ replaces it regularly (e.g., once a week). XZDH provides online deniability, the same forward secrecy characteristics as X3DH, and the same offline deniability as ZDH—anyone can forge an XZDH transcript between $\mathcal{I}$ and $\mathcal{R}$ given only g^I, g^R, and a prekey bundle containing g^{Γ} and $\text{Sig}(g^I, I, g^{\Gamma})$. Since these values are all publicly distributed, XZDH provides much stronger offline deniability than X3DH.

Proposition 4 states a simplified security theorem for XZDH.

Proposition 4 | XZDH is secure (informal)

If the MAC is weakly unforgeable under chosen message attack [BKR00], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then XZDH provides the security properties listed in Section 5.2.2 against adaptive corruptions when the response is sent by an honest party. (Refs: 101 and 502)

A formal theorem and proof sketch appear in Chapter 6.

5.7 Secure Messaging Integration

This section covers several practical considerations involved in securely implementing DAKEZ, Spawn$^+$, ZDH, and XZDH for use in secure messaging tools.

All elements in the ciphertexts must be encoded using a method that is unambiguously parseable (e.g., using length-prefixed strings and fixed-length integer encodings). The identifiers for the parties (e.g., "I" and "R") may be cryptographic hashes of usernames, or they may include the long-term public keys. Identifiers can also include device codes to differentiate between devices owned by a user. In general, the problem of supporting multi-device conversations is difficult and mostly orthogonal. Any multi-device technique that works for 3DH or X3DH should also be applicable to the new DAKEs; the most common techniques in practice—replicating one key or generating per-device keys [AH16]—are directly applicable.

While the definitions in this chapter are given in terms of generic groups, the primitives should be implemented using elliptic curve cryptography for maximum performance. Parties must also ensure that the DH contributions they receive are in the expected group and are not the identity element. Otherwise, the implementation may be vulnerable to identity-element [Din05], small-subgroup [LL97], or invalid-curve [ABM+03] attacks.

As discussed in Section 5.5, interactive DAKEZ and Spawn$^+$ provide online deniability for both $\mathcal{I}$ and $\mathcal{R}$, while ZDH and XZDH do not. Consequently, for applications in which interactive communication is *sometimes* possible, developers should consider a hybrid approach where interactive DAKEZ is attempted first, and non-interactive ZDH or XZDH is used as a fallback option. This hybrid approach enables non-interactive messaging and minimizes the use of one-time prekeys, but sacrifices online deniability for $\mathcal{R}$ when $\mathcal{I}$ is willing to use the fallback option.

Using ZDH or XZDH non-interactively requires an untrusted central server to store prekeys. If prekey submissions are not authenticated, then malicious users can perform denial-of-service attacks. To preserve the deniability of the overall protocol, one-time prekeys should never be digitally signed. The best approach is to authenticate prekey uploads using a DAKEZ exchange between the uploader and the server, which preserves deniability. As an added safeguard, the server can require a zero-knowledge proof of knowledge of the private keys associated with the prekeys.

All of the new DAKEs permit authenticating the shared session state Φ as part of the exchange. Theoretically, this ensures that the DAKE is "bound" to a session of the messaging protocol. The higher-level protocol should always include its implicit or explicit session identifier in Φ to perform this binding. In practice, Φ also allows both sides to cryptographically verify some beliefs they

have about the session. For example, in an application that assigns some attribute to users before a conversation (e.g., a networked game in which players take on specific roles), the expected attributes (encoded in an unambiguously parseable manner) should be included in Φ. If the DAKE succeeds, then the participants know that they agree about the attribute values.

Implementations of the DAKEs should follow standard secure coding practices for cryptographic software. Cryptographic operations with secret data should be performed in constant time to mitigate side-channel attacks. Secret values, including intermediaries, should be stored in locked private memory pages, and ephemeral keys should be securely erased in a manner guaranteed to persist through compiler optimizations. Hash functions used for different purposes should be domain separated using application-specific strings.

Deployments of the DAKEs should always include tools to forge key exchanges; this improves plausible deniability in practice. Specifically, implementations of RSig from Section 5.3.3 should use the same code for honest and forged authentication. To accomplish this in constant time, developers can use conditional move operations to copy the appropriate keys into memory regions for the calculations, and again to move the results into the appropriate positions in the proof.

5.7.1 Incorporating Quantum Resistance

Selecting a quantum-resistant KEM to use in an implementation is challenging. Recently, dozens of schemes have been developed and reviewed as part of the NIST standardization effort [AAA+20]. These schemes offer a wide variety of performance and security characteristics, and there is no protocol that is clearly superior to all others. Fortunately, the generality of the modular quantum KEM functions defined in Section 5.3.4 allows many schemes, such as the eventual winners of the NIST standardization process, to be seamlessly integrated into the DAKEs. Developers should consider computational requirements, key size, ciphertext size, the security assumptions, and the probability of decapsulation failure when selecting a quantum-resistant KEM.

As mentioned in Section 5.3.4, the security proofs for the new DAKEs require that if a quantum-resistant KEM is used, it must provide quantum IND-CPA security, and it must be $2^{-\lambda}$-correct (or better) for security parameter λ. Almost all of the finalists and alternate candidates in the third round of the NIST standardization effort satisfy these requirements for $\lambda = 128$.[14]

[14] All of the proposed schemes are either correct or δ-correct for $\delta \leq 2^{-128}$ with the exception of SABER, which is only 2^{-120}-correct when using the parameter set for 128-bit security. It is reasonable to consider this failure probability to be effectively negligible. All of the proposed schemes provide quantum IND-CPA security, with many additionally providing quantum IND-CCA2 security.

Historically, some quantum-resistant KEMs, such as New Hope [ADPS16], only provided δ-correctness for a much larger δ (e.g., $\delta = 2^{-60}$). This practice has been effectively abandoned after it became apparent that decapsulation failures are typically a promising avenue for cryptanalysis [GJY19]. Developers should ensure that a quantum KEM satisfies the security and correctness requirements before using it with the new DAKEs. Beyond these basic requirements, developers should consult the results of the NIST standardization effort to compare performance and security characteristics of popular quantum-resistant KEMs.

5.8 Implementing the Protocols

Although many DAKEs have been described in the literature, very few of them have ever been implemented. If a DAKE is designed with the intention of protecting real users, then producing a proof-of-concept implementation has several benefits. In addition to enabling experiments to quantify real-world performance, the implementation process often reveals practical hurdles limiting deployability that cannot be identified from a detached security analysis, necessitating improved designs. For these reasons, a prototype implementation of the new DAKEs was developed[15] in C using production-ready cryptographic libraries targeting a 128-bit security level. The implementation employs all of the secure coding techniques mentioned in Section 5.7. The prototype code and evaluation data are available at crysp.uwaterloo.ca/software.

The remainder of this section provides implementation details. Section 5.8.1 lists the third-party libraries used in the prototype, Section 5.8.2 describes the additions to the elliptic curve library that were necessary, and Section 5.8.3 lists the chosen primitives.

5.8.1 Libraries

The implementation uses version 1.1.0e of the OpenSSL library for its cryptographic randomness source and its AEAD primitives. The implementation uses the twisted Edwards curve associated with Ed25519 [BDL+12] for its group operations. The `ed25519-donna` library [Moo12] with SSE2 extensions was selected due to its speed and portability. Curve points generated by the library are packed into 32 bytes, which includes the y-coordinate and a parity bit of the x-coordinate, for transmission and storage. When unpacked, the points can be multiplied by

[15] The prototype implementation described in this section was developed in 2017. While many of the dependencies have released updates in the intervening years, the implementation still accomplishes its objectives: acting as a reference implementation of the new DAKEs, and facilitating a performance evaluation.

the subgroup order and compared to the identity element to ensure that they are in the correct subgroup and not on the twist, although this is not required for the security of Ed25519 signatures [BDL+12]. The eXtended Keccak Code Package [Van13a] provides the SHA-3 implementation for the prototype. To hash curve points, the packed 32 bytes are hashed with SHA-3.

5.8.2 Ed25519-donna Additions

Although Curve25519 [Ber06] is used in Signal [Ope13] and is generally well-regarded in the secure messaging community, existing libraries cannot be used to implement the new DAKEs without modification. X25519 [Ber06] libraries typically store points in Montgomery form, which discards the sign and complicates ElGamal implementations. Ed25519 [BDL+12] libraries store points on a birationally equivalent twisted Edwards curve, but do not provide independent point addition or scalar multiplication functions. Consequently, implementing the prototype required several additions to `ed25519-donna`.

The prototype implements constant-time scalar multiplication with variable bases using the windowed double-and-add-always algorithm. To select random scalars, the prototype uses the X25519 secret generation procedure [Ber06]. Existing library code was adapted to produce constant-time implementations of point negation, scalar subtraction, conditional memory copies, and scalar equality tests.

5.8.3 Primitive Instantiations

To instantiate the DREAD scheme from Section 5.3.2, the prototype uses OpenSSL's implementation of AES256-OCB with a 16-byte tag and a 15-byte nonce. The first 15 bytes of the L hash derived in the NIZKPK are used as the nonce for OCB mode, since its probability of repeating is negligible. The implementation computes the secret group element by multiplying the Ed25519 base point by the random scalar $\mathcal{K}$.[16] To derive the AEAD key, the secret group element is packed into its 32-byte representation and hashed using SHA3-256 as a KDF. SHA3-256 is also used to instantiate the hash H used to compute c in RSig.

The prototype supports arbitrary caller-defined user identifiers and shared session state Φ. The ZDH and XZDH implementations use KMAC256 [KCP16] as the MAC, since it is based on SHA-3. Although all of the hash input fields are of fixed length, the prototype implements the

[16] This is not a safe method for selecting a random point in general, since the discrete log of the point is known. The method is safe for this application because the result is only input into a KDF.

complete KMAC256 padding scheme to reflect the overhead of using general-purpose KMAC libraries.

5.9 Performance Characteristics

When deciding whether or not to include a DAKE in a secure messaging application, one of the most important considerations for developers is the expected performance impact on both users and infrastructure. This section provides an analysis of the prototype implementation to precisely quantify this impact by answering the following questions for DAKEZ, Spawn$^+$, ZDH, and XZDH:

- What is the computational overhead for generating long-term keys and participating in key exchanges?
- How large are the long-term public keys?
- How large are the prekeys (where applicable)?
- How much data is transferred during a key exchange?
- How does the performance of the new DAKEs compare to other state-of-the-art AKEs?

The performance of the new DAKEs was compared to four AKEs with weaker security properties—Elliptic Curve Diffie-Hellman (ECDH), 3DH [Mar13], X3DH [MP16], and SIGMA-R [Kra03]—and the three previously known DAKEs with the same properties—Φ_{idre}, RSDAKE, and Spawn [UG15]. (Φ_{idre} [UG15] is a more efficient instantiation of Φ_{dre} [DKSW09] using an interactive multi-round DRE construction.)

3DH, X3DH, ZDH, and XZDH are all designed for non-interactive messaging (e.g., text messaging). ZDH and XZDH use ROM-based ring signatures to achieve strong deniability, which 3DH and X3DH lack. Φ_{idre}, RSDAKE, Spawn, and DAKEZ are all designed for interactive communication. DAKEZ is similar to RSDAKE, except that it avoids the need for digital signatures, and uses ROM-based ring signatures. Φ_{idre} provides strong deniability using DRE in the standard model, but it requires an excessive number of network round trips to do so.

In order to fairly compare the high-level designs, ECDH, 3DH, X3DH, and SIGMA-R were implemented using the primitives described in Section 5.8: group operations over the Ed25519 twisted Edwards curve, SHA3-256 for hashing and key derivation, and KMAC256 for message authentication. X3DH and XZDH prekeys were signed with EdDSA. SIGMA-R was implemented

Table 5.1 PERFORMANCE EVALUATION OF NEW DAKES. (Refs: 107 [a] [b])

	ECDH	3DH	X3DH	SIGMA-R	Φ_{idre}	RSDAKE	Spawn	DAKEZ	Spawn$^+$	ZDH	XZDH
Offline Deniable	●	●	○	○	●	●	●	●	●	●	●
Online Deniable	●	-	-	-	●	●	○	●	○	○	○
Authenticated	-	●	●	●	●	●	●	●	●	●	●
Non-Interactive	●	●	●	-	-	-	●	-	●	●	●
Forward Secrecy	-	-	○	●	●	●	-	●	-	-	○
Proof Model	SM	ROM	ROM	ROM	SM	SM	SM	ROM	ROM	ROM	ROM
Key Generation [ms]	-	0.0228 (0.0012)	0.0240 (0.0013)	0.0240 (0.0012)	0.40 (0.01)	206 (8)	206 (4)	0.0440 (0.0016)	0.0429 (0.0016)	0.0441 (0.0018)	0.0444 (0.0017)
Exchange [ms]	0.1733 (0.0033)	0.4229 (0.0050)	0.5533 (0.0056)	0.3478 (0.0048)	13 (2)	6 630 (50)	3 390 (20)	1.094 (0.014)	1.3683 (0.0082)	0.778 (0.013)	0.9217 (0.0069)
Flows	2	2	2	4	9	3	2	3	2	2	2
Public Key [B]	-	32	32	32	415	395	992	32	32	32	32
Prekey [B]	-	32	32+96	-	-	-	938	-	32	32	32+96
Exchange [B]	64	80	80	272	5 140	7 598	73 763	464	512	304	304

● = provides property; ○ = partially provides property; - = does not provide property / not applicable; SM = standard model; ROM = random oracle model. Standard deviations are in parentheses. "Forward secrecy" is the strong variant [BPR00] (all schemes have weak forward secrecy). Prekeys are listed as (one-time)+(signed) sizes.

as described by Di Raimondo et al. [DGK05] using EdDSA for signatures, and without the identity protection mechanism in the OTR protocol [AG07]. Φ_{idre}, RSDAKE, and Spawn were implemented using the primitives selected in the publication that introduced them [UG15, §7], which are secure in the standard model. The analysis used 8-byte identifiers for users, and no higher-level shared session data. It also included identifiers in the transmission costs for non-interactive protocol, since this reflects real deployments.

The key exchanges were performed using all 11 protocols on one Intel Skylake core pinned to 4.0 GHz with 8 MB of L3 cache and Intel Turbo Boost disabled. For each protocol, the analysis measured the thread-specific CPU time required to perform a key exchange and, independently, to generate a long-term public key. A total of 100,000 measurements were collected for each protocol except for Φ_{idre}, RSDAKE, and Spawn, where the computational costs restricted the analysis to 1,000 measurements each. The size of the long-term public keys, prekeys (where applicable), and the total key exchange traffic (including one-time prekeys) was also measured for each protocol. Table 5.1 compares the schemes and presents the results. The "partial" forward secrecy for X3DH and XZDH in the table denotes the characteristics described in Section 5.6.

Table 5.1 shows the dramatic difference between the protocol classes. The schemes without authentication and strong deniability (left) require less than 1 ms to generate long-term keys or complete key exchanges. In contrast, the strongly deniable schemes in the standard model [UG15] (middle) are prohibitively more expensive: RSDAKE and Spawn require several *seconds* for exchanges and require kilobytes of transmission, while Φ_{idre} requires over four network round trips. Despite the computational efficiency of Φ_{idre}, it is often the slowest protocol in practice

due to network round-trip time dominating all other costs in typical networks. DAKEZ, Spawn$^+$, ZDH, and XZDH (right) provide authentication and strong deniability while nearly matching the weaker schemes' performance: the new DAKEs require roughly 1 ms of CPU time and only slightly more communication than SIGMA-R. The one-time prekeys are 32 bytes for all non-interactive schemes except Spawn, which uses 938 bytes. The signed prekeys for X3DH and XZDH are 96 bytes.

The results show that DAKEZ, ZDH, and XZDH are the first schemes to offer strong deniability with only slightly more overhead than AKEs used in popular secure messaging protocols. While comparing Spawn to Spawn$^+$ demonstrates the value of ROM-based primitives, DAKEZ, ZDH, and XZDH are all more efficient. For non-interactive settings, XZDH provides better forward secrecy than ZDH with very small overhead.

5.10 Key Compromise Impersonation Attacks

One aspect of online deniability that is often overlooked in the literature is the relationship between strongly deniable key exchange protocols and key compromise impersonation attacks. A KCI attack begins when the long-term secret key of a user of a vulnerable DAKE is compromised. With this secret key, an adversary can impersonate *other* users *to* the owner of the key. DAKEs offering online deniability, such as Φ_{idre}, RSDAKE, Spawn, DAKEZ, Spawn$^+$, ZDH, and XZDH, are inherently vulnerable to key compromise impersonation attacks. Moreover, "vulnerability" in this context is actually a desirable property.

In theory, a user who claims to cooperate with a judge may justifiably refuse to reveal their long-term secret key because it would make them vulnerable to a KCI attack. The design of strongly deniable DAKEs makes it impossible for the user to provide proof of communication to the judge without also revealing their long-term secret key. This is the primary benefit and motivation of this class of key exchanges: they prevent a judge and informant from devising a protocol wherein the judge is given cryptographic proof of communication while the informant suffers no repercussions.

However, this scenario may be mostly theoretical. The more common case in practice may be the one in which the judge has access to the user's long-term secret keys. A typical real-world example is when a user is forced to surrender and unlock their mobile device with a secure messaging application installed; American and Canadian border agents currently exercise this power over travelers [Gru17; Gol15]. In this situation, the KCI "vulnerability" also becomes an asset. This section uses non-interactive Spawn$^+$ as an example for simplicity, but the ideas can be extended to the other DAKEs.

5.10.1 Attacks Against Spawn$^+$

The security of Spawn$^+$ does not require trusting the central server used to distribute prekeys. However, in a scenario where the user's keys have been compromised but the central server has not, it is possible to achieve better plausible deniability. The user may ask the central server in advance to assist with a forged conversation, casting doubt on all conversations conducted by the judge using the compromised device.

If the judge attempts to act as $\mathcal{I}$ in a conversation using the compromised device, then the user (or a trusted accomplice with access to the long-term secret I) can impersonate $\mathcal{R}$ by executing RSig with g^I and I instead of g^R and R. In practice, the user (or accomplice) simply needs to run the protocol honestly, but pretend to be $\mathcal{R}$ in their response to the prekey.

If the judge attempts to act as the responder $\mathcal{R}$ of a conversation using the compromised device, then the situation can be somewhat improved, but it is not possible offer full deniability. The user must ask the central server to return a false prekey for $\mathcal{I}$ that was generated by the user or their trusted accomplice, and to redirect all traffic to the associated forging device. This false prekey must be returned to the judge when they request one. The user can derive the shared secret k by decrypting the DREAD ciphertext using I instead of R. In practice, the judge can always bypass this forgery attempt by obtaining a legitimate prekey for $\mathcal{I}$ and using this to respond using R. This is a fundamental limitation of Spawn$^+$ that also applies to Spawn, and is conjectured to be insurmountable by a two-flow non-interactive protocol [UG15]. The design of ZDH in Section 5.5 explicitly acknowledges this limitation to improve performance.

5.10.2 Limiting or Preventing Attacks

If the KCI vulnerability is undesirable, it is possible to make all of the new DAKEs more resilient to it while maintaining their deniability properties. To do so, a protocol like Spawn$^+$ can be altered to include long-term "forger" keys for all participants. For example, initiator $\mathcal{I}$ would distribute both g^I and g^{F_I} as its public key, where g^{F_I} is $\mathcal{I}$'s public forging key, and F_I is the associated secret key. The second flow in the Spawn$^+$ key exchange is then altered so that the ciphertext and proof are computed as follows: $\gamma = \text{DREnc}(g^I, g^{F_R}, g^r \| Q_R, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I)$, and $\sigma = \text{RSig}(g^R, R, \{g^{F_I}, g^R, g^i\}, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I \| \gamma \| \Phi)$.

In general, this transformation changes all long-term public keys in the protocol that are not used in the "honest" case to reference the forging keys instead. This alteration allows the forging keys to be stored more securely than the "honest" public keys; since the forging keys are not needed for normal operation, they may be stored offline (e.g., on paper in a vault). Alternatively, if a user (or developer) is more concerned about preventing KCI attacks than providing online

deniability, the forging secret keys can be destroyed immediately after generation; this will sacrifice online deniability for the user, but also prevent KCI attacks against them.

Counterintuitively, implementing this option for users can actually provide both benefits in practice. Consider a secure messaging application that asks users whether or not they would like to save forging keys during setup. Even if most users select the default option to securely erase the forging keys, thereby preventing them from performing the online forgery techniques described above, a judge does not generally know the choice of a particular user. Consequently, a judge that engages in a conversation using a compromised device is given two explanations: either the conversation is genuine, or the owner of the device was one of the users that elected to store the forgery keys and they are using those keys to forge the conversation. The result is that a degree of plausible deniability is preserved, even though most users in this scenario become immune to KCI attacks.

The same general transformation works for the other DAKEs described in this chapter. Note that trust establishment (e.g., physical exchange of key fingerprints) must cover both keys in this scheme. This is most easily accomplished by verifying a fingerprint derived from a hash of both keys.

5.11 Chapter Summary

This chapter introduced three new two-party DAKEs—the first to practically provide strong deniability for secure messaging applications in both interactive and non-interactive settings. Section 5.1 motivated the need for strong deniability by describing insider attacks against OTRv3 and Signal that are only possible because these protocols use DAKEs without online deniability. Section 5.2 described the desired security properties for the new DAKEs. Section 5.3 defined and constructed the necessary cryptographic primitives: Section 5.3.2 defined DREAD, which adds authenticated associated data to DRE cryptosystems; Section 5.3.2 also presented a DREAD construction built using hybrid ElGamal encryption and a NIZKPK of plaintext equality; Section 5.3.3 presented an efficient three-element ring signature scheme in the ROM constructed using a Schnorr-based NIZKPK; and Section 5.3.4 defined the interface for a quantum-resistant KEM in order to provide quantum transitional security in the new DAKEs. Section 5.4 introduced DAKEZ, a new DAKE meant for interactive secure messaging settings like instant messaging. Section 5.5 introduced ZDH (and a ROM-based variant of Spawn that is useful for comparison purposes), a new DAKE meant for non-interactive settings that is a drop-in replacement for 3DH. Section 5.6 introduced XZDH, which improves the forward secrecy of ZDH in the same way that X3DH improves the forward secrecy of 3DH. Section 5.7 provided practical advice and covered

common security pitfalls for developers choosing to implement the new DAKEs. Section 5.8 described a prototype implementation of the new DAKEs and competitive schemes constructed using the same primitives, and Section 5.9 analyzed this prototype as part of a performance evaluation. Finally, Section 5.10 described the relationship between online deniability and KCI attacks, as well as a technique to mitigate KCI attacks while preserving plausible deniability.

While this chapter introduced powerful new DAKEs, it focused heavily on a high-level overview and technical details that are primarily beneficial to developers. In particular, the security properties of the DAKEs were presented in a very informal manner for clarity of presentation. However, since AKEs have been historically prone to security oversights, it is important to provide a formal proof of the claimed security properties whenever possible. The next chapter presents an in-depth security analysis of the new DAKEs presented in this chapter, and proves that they provide their security guarantees using simulation-based security proofs.

CHAPTER 6 | Proving the Security of DAKEZ, Spawn$^+$, ZDH, and XZDH

In this chapter: Σ Proof

Three new two-party DAKEs with strong deniability intended for secure messaging applications were introduced in Chapter 5: DAKEZ (Section 5.4), ZDH (Section 5.5), and XZDH(Section 5.6). Chapter 5 informally defined the security properties of the new DAKEs and proved the security of the underlying DREAD and RSig/RVrf primitives, but it did not provide formal security proofs for the DAKEs. This chapter formally defines the threat model and security properties for the new DAKEs and sketches security proofs for all of these properties. These proof sketches are academically interesting contributions in their own right, as they include the definition of several components and techniques that can be reused for the development and analysis of future DAKEs. However, the details in this chapter are highly technical and will only be of interest to cryptographers; readers who are uninterested in the minutia of security proofs may wish to skip to Chapter 7.

Section 6.1 outlines the general approach to prove the security of the DAKEs and the necessary preliminaries. The security proofs for DAKEZ, Spawn$^+$, ZDH, and XZDH are sketched in Sections 6.2, 6.3, 6.4, and 6.5, respectively. Although Spawn$^+$ is an illustrative construction that is not intended to be used in practice, proving its security is useful for establishing techniques that are reused when proving ZDH's security.

6.1 DAKE Security Proof Techniques Σ

Several techniques have been suggested for modeling and proving deniability properties of DAKEs. Di Raimondo et al. first formalized the notion of DAKEs [DGK06]. More recently, Fischlin and Mazaheri [FM15] proposed weaker deniability notions that can characterize the properties of SIGMA, 3DH, and X3DH. Dodis et al. [DKSW09] pointed out that the notions of online and offline deniability have natural parallels in security proofs within the universal composability

(Universal Composability (UC)) framework introduced by Canetti [Can01]. UC is a framework for instantiating security models and using them to prove the security of protocols. A protocol that is secure in a UC-based model is guaranteed to retain its security properties under arbitrary compositions, even when arbitrary protocols are run concurrently.

A UC security proof involves defining an *ideal functionality* that describes a protocol with self-evident security properties. This ideal functionality is executed by a trusted authority that protocol participants, and an adversary, interact with. The functionality defines the possible interactions between the parties and the authority; in the case of the adversary, messages sent to the authority model adversarial control over protocol execution, and messages sent to the adversary represent information disclosures. A UC security proof involves showing that any adversary attacking a particular "real" protocol (without a trusted authority) can be used to construct an attack against the ideal functionality, thereby showing that their security properties are equivalent. A protocol with this property is said to *UC-realize* the functionality. To show this property, proofs demonstrate that no external environment can distinguish between an adversary interacting with the real protocol and a simulator interacting with the ideal functionality. This environment can communicate with the adversary (or simulator), control the inputs of the protocol participants, and read their outputs, but it cannot directly view or interact with messages exchanged between the participants. In contrast, the adversary is given complete control over messages transmitted between participants. The adversary (or simulator) can also corrupt parties, which provides complete control over their future interactions, and reveals all memory state that has not been erased (if memory erasure is permitted by the model). The external environment is notified when parties are corrupted.

Internally, the computational model of UC is defined in terms of interactive Turing machines with "secure" and "insecure" tapes, and the ability to invoke other machines as subroutines. The proof sketches in this chapter are not expressed in terms of these internal modeling details; interested readers are referred to Canetti's definitions for the formalization [Can01]. The proof sketches in this chapter assume that the reader is familiar with the UC framework.

Canetti et al. defined the generalized universal composability (GUC) framework for the purpose of proving strong deniability properties [CDPW07]. The GUC framework is an extension of UC in which all machines (including the ideal functionality and the external environment) are granted access to *shared functionalities* that persist between protocol sessions. This provides a natural way to model features like a PKI. The security proof sketches for RSDAKE and Spawn [UG15] also used the GUC framework.

6.1.1 The GUC Framework

Following Dodis et al. [DKSW09] and Unger and Goldberg [UG15], the security proofs for the new DAKEs are sketched in the GUC framework. The GUC framework corrects a deficiency in the basic UC framework: in many protocols, some state information persists across sessions (e.g., long-term public keys). This information should be available to the external environment, because it could be used to distinguish between simulators and real protocols. This persistent state is captured by shared functionalities.

For maximum usefulness, the new sketches rely on traditional game-based security proofs for the primitives (i.e., the DREAD and ring signature schemes) within the GUC framework; this makes it easy to substitute primitives with other well-known constructions, while also providing strong composability guarantees for the new DAKEs. The new sketches use a GUC-based security model including random oracles.

Security proofs in the GUC framework follow four steps: define an ideal functionality with the desired security properties, define a "real" protocol, construct a simulator $\mathcal{S}$ attacking the ideal functionality based on an adversary $\mathcal{A}$ attacking the real protocol, and show that the two scenarios are externally indistinguishable given the security assumptions. The main participants in a protocol are called the *principal parties*. In the real setting, these parties use their inputs to exchange messages over a network controlled by $\mathcal{A}$, then generate outputs. In the ideal setting, these parties are *dummy parties* that simply forward inputs and outputs to and from the ideal functionality over a secure channel.

To simplify the process of producing GUC security proofs, Canetti et al. [CDPW07] introduced the External-subroutine Universal Composability (EUC) framework. EUC is equivalent to GUC, except that it constrains the scheme to a single shared functionality $\bar{\mathcal{G}}$ (an ideal functionality meeting this definition is called $\bar{\mathcal{G}}$*-subroutine respecting*). Moreover, in the EUC framework it is only necessary to consider a single session of the challenge protocol. Canetti et al. [CDPW07] proved that EUC security is equivalent to GUC security for $\bar{\mathcal{G}}$-subroutine respecting protocols [CDPW07, Th. 2.1], greatly reducing the complexity of proofs. Notably, this surprising result means that a $\bar{\mathcal{G}}$-subroutine respecting protocol that *EUC-realizes* an ideal functionality also *GUC-realizes* that ideal functionality when it is run concurrently with itself and with arbitrary external protocols. This results enables the new proof sketches to use the simpler EUC framework, while the results can be extended to the GUC framework.

6.1.2 Proof Notation and Setup

The new proof sketches adopt standard notational conventions for interactive Turing machines in the EUC/GUC framework. Ideal functionalities include $\mathcal{F}$ in their name. The external environment, denoted by $\mathcal{Z}$, attempts to distinguish between a simulator $\mathcal{S}$ attacking the ideal functionality and a real adversary $\mathcal{A}$ attacking a real protocol. $\overline{P}$ denotes an idealized party interacting with an ideal functionality in the EUC/GUC framework; $\mathcal{P}$ denotes the corresponding party in a real protocol. The corresponding party simulated by $\mathcal{S}$ for $\mathcal{A}$ is denoted as $P^{(s)}$. As a notational convenience, $\overline{P}$ is written within the context of a EUC/GUC message to denote a label for the party. Similarly, $\mathcal{P}$ is written in the context of a real protocol message to denote the same label for the party.

The GUC framework permits more general interactions between entities, but due to the aforementioned equivalence of the frameworks, the sketches can focus solely on EUC interactions while effectively producing GUC proofs. In summary, in the EUC framework, $\mathcal{Z}$ is permitted to securely communicate with $\mathcal{A}$ (or $\mathcal{S}$), control the inputs of every principal party $\mathcal{P}$ (or $\overline{P}$), and read their outputs. $\mathcal{A}$ is given control over the interactions of every principal party $\mathcal{P}$ (in the real setting) or $P^{(s)}$ (when being simulated by $\mathcal{S}$), while $\mathcal{S}$ is permitted to interact with the ideal functionality in the prescribed manner. Both $\mathcal{A}$ and $\mathcal{S}$ are able to corrupt principal parties, and these corruptions are reported to $\mathcal{Z}$. The ideal functionality, $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{Z}$ are all permitted to interact with with the shared functionality in the prescribed manner.

In the new security proof sketches, code for interactive Turing machines is given in event-based C-like pseudocode for clarity. The **return** keyword indicates that message processing immediately ceases; **return** combined with **if** expresses publicly known constraints on message values. The sketches adopt the common notion of "delayed messages" to mean that the ideal functionality gives $\mathcal{S}$ control over the timing and success of the message delivery by sending a message to $\mathcal{S}$, and delivering the original message only upon receipt of a delivery instruction message from $\mathcal{S}$.

The execution of a system of interactive Turing machines takes place sequentially (i.e., only one machine is active at any point in time) [Can01]. The new proof sketches follow the execution semantics defined by Canetti and Krawczyk for real protocols [CK02b, Fig. 1] and ideal processes [CK02b, Fig. 2], which unambiguously specify the activation order of machines. Note that these semantics allow principal parties in real protocols to send one message and locally output a value in the same activation (after which $\mathcal{Z}$ becomes active), and they also allow ideal functionalities to send multiple messages in an activation (if the functionality sends a message to $\mathcal{S}$ then $\mathcal{S}$ becomes active, otherwise the previously activated party becomes active).

$\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$ denotes the shared functionality depicted in Algorithm 6.1, which is used by all of the new proof sketches. This shared functionality models two types of cross-session state: the registration and distribution of long-term public keys, and a collection of domain-separated random oracles. This is accomplished by merging together two shared functionalities that were previously defined in the literature: $\bar{\mathcal{G}}_{krk}^{\varphi}$ and $\bar{\mathcal{G}}_{ro}$. $\bar{\mathcal{G}}_{krk}^{\varphi}$ refers to the *key registration with knowledge* PKI shared functionality defined by Dodis et al. [DKSW09, Fig. 2], which distributes public keys to all parties, but reveals the corresponding secret keys only to corrupted owners and the ideal functionality $\mathcal{F}$. $\bar{\mathcal{G}}_{ro}$ refers to the random oracle shared functionality defined by Walfish [Wal08, Fig. 2.4]. $\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$ essentially combines $\bar{\mathcal{G}}_{krk}^{\varphi}$ with n copies of $\bar{\mathcal{G}}_{ro}$ in order to explicitly model the notion of domain separation. It is necessary to combine these functionalities into one so that the EUC framework applies.

$\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$ is parameterized with a set of interactive Turing machine programs, φ, that are permitted to retrieve secret keys.[1] Corrupt parties can always retrieve their own secret keys. The new sketches permit secret keys to be retrieved by honest parties running the real key exchange protocol (so that they can perform the protocol without revealing their secret keys to $\mathcal{Z}$) and by the ideal functionality. This restricts honest parties to safe usage of their long-term secret keys. Borrowing a notational convenience from Walfish [Wal08, §3.3], $\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$ is implicitly parameterized with the ideal functionality that a proof sketch is attempting to realize.

Note $\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$ does not provide the ability to "reprogram" random oracle results or to extract random oracle queries. This differs from traditional random oracle models, but follows the GUC-based functionality defined by Walfish [Wal08]. The main reason for omitting these features is that simulators in the UC (and EUC/GUC) framework must be straight-line simulatable, since $\mathcal{S}$ is given only black-box access to $\mathcal{Z}$ and has no control over it [Lin03]. This is an intuitive result, since $\mathcal{Z}$ effectively represents concurrent protocols, and these protocols have direct access to the same random oracle. In general, $\mathcal{Z}$ can perform random oracle queries on its own, and transfer the results to $\mathcal{A}$ or $\mathcal{S}$ to hide the queries and avoid the possibility of reprogramming. However, indistinguishability proofs have no such restrictions, and reductions may rewind $\mathcal{Z}$ or reprogram the random oracle, since these reductions can *internally* execute $\mathcal{Z}$ and $\mathcal{S}$. This technique can be used to employ the Fiat-Shamir heuristic in protocols within the GUC framework by performing the standard reduction [BR93b] involving programming the random oracle [Wal08, Th. 5.10]. All of the simulators in this chapter are straight-line simulatable and these simulators do not reprogram the random oracle. The simulators do not require the extraction of witnesses from

[1] In the UC framework (and by extension in the GUC and EUC frameworks), machines can inspect the code being run by other machines to see if they are running a particular program. The shared functionality uses this technique to ensure that only the intended ideal functionality can retrieve uncorrupted secret keys, enabling a simple analysis of the key secrecy property.

Algorithm 6.1 SHARED FUNCTIONALITY FOR KEY REGISTRATION WITH KNOWLEDGE AND RANDOM ORACLES. (Refs: 116 and 160)

Shared Functionality $\bar{\mathcal{G}}_{krkro}^{\varphi,n,\mathbb{G},q,g}$

Parameterized by an implicit security parameter λ, a set of interactive Turing machine programs φ, a number of random oracles n, and a group $\mathbb{G}$ generated by g with prime order q.

on $(\texttt{register})$ **from** $\mathcal{P}$:
  if ($\mathcal{P}$ is corrupt) **return**
  if (there is a record $(\texttt{key}, \mathcal{P}, \cdot, \cdot)$) **return**
  $SK \xleftarrow{\$} \mathbb{Z}_q$
  $PK \leftarrow g^{SK}$
  Record $(\texttt{key}, \mathcal{P}, PK, SK)$

on $(\texttt{register}, SK)$ **from** $\mathcal{P}$:
  if ($\mathcal{P}$ is not corrupt) **return**
  if (there is a record $(\texttt{key}, \mathcal{P}, \cdot, \cdot)$) **return**
  $PK \leftarrow g^{SK}$
  Record $(\texttt{key}, \mathcal{P}, PK, SK)$

on $(\texttt{retrieve}, \mathcal{Q})$ **from** $\mathcal{P}$:
  if (there is a record $(\texttt{key}, \mathcal{Q}, PK, SK)$) {
    Send $(\texttt{pubkey}, \mathcal{Q}, PK)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{pubkey}, \mathcal{Q}, \perp)$ to $\mathcal{P}$
  }

on $(\texttt{retrievesecret}, \mathcal{Q})$ **from** $\mathcal{P}$:
  if (($\mathcal{P}$ is honest) $\wedge$ ($\mathcal{P}$'s code $\notin \varphi$)) {
    return
  }
  if (($\mathcal{P}$ is corrupt) $\wedge$ ($\mathcal{P} \neq \mathcal{Q}$)) **return**
  if (there is a record $(\texttt{key}, \mathcal{Q}, PK, SK)$) {
    Send $(\texttt{seckey}, \mathcal{Q}, PK, SK)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{seckey}, \mathcal{Q}, \perp, \perp)$ to $\mathcal{P}$
  }

on $(\texttt{ro}, i, x)$ **from** $\mathcal{P}$:
  if ($i \notin [1, n]$) **return**
  if (there is a record $(\texttt{ro}, i, x, v)$) {
    Send $(\texttt{ro}, v)$ to $\mathcal{P}$
  } **else** {
    $v \xleftarrow{\$} \{0,1\}^\lambda$
    Record $(\texttt{ro}, i, x, v)$
    Send $(\texttt{ro}, v)$ to $\mathcal{P}$
  }

NIZKPKs or SoKs, and thus the protocols do not require straight-line extractable zero-knowledge proofs (e.g., Ω-protocols [GMY03]).

While this chapter does not include explicit reductions in the proof sketches, the indistinguishability proofs note when indistinguishability depends on the security assumptions of the primitives. In these cases, it is easy to construct reductions that attack the assumptions by internally executing $\mathcal{S}$ and a $\mathcal{Z}$ with distinguishing advantage, programming values as necessary to insert the reduction question, and then using the results of $\mathcal{Z}$ to complete the attack.

6.2 Proof of DAKEZ Security (Σ)

To prove the security of DAKEZ within the GUC framework, an ideal functionality that represents a DAKE with the desired features and security properties must be selected. Unfortunately, the ideal functionalities defined by Dodis et al. [DKSW09] ($\mathcal{F}_{keia}^{\text{IncProc}}$) and Unger and Goldberg [UG15] ($\mathcal{F}_{post-keia}^{\text{IncProc}}$) do not capture all of the desired properties, so a new functionality is required. Section 6.2.1 introduces and discusses this new functionality. Section 6.2.2 formally defines DAKEZ with an interface matching the ideal functionality. Section 6.2.3 presents the security theorem and an overview of the proof strategy. Section 6.2.4 relates the protocol properties in Section 5.2.2 to the ideal functionality definition. Finally, Section 6.2.5 describes the actual proof sketch and subsequent sections of this chapter.

6.2.1 Ideal Functionality for DAKEZ

6.2.1.1 Contributiveness

In their original analysis of universally composable key exchanges, Canetti and Krawczyk [CK02b] proposed an ideal functionality, $\mathcal{F}_{ke}$, that sends a randomly selected session key to the participants. They noted that this functionality cannot be realized by protocols like two-flow DH, since the simulator must "commit" to a shared secret without knowledge of the session key selected by the ideal functionality. Non-static adversarial corruptions can distinguish between real and ideal protocols by corrupting ephemeral state prior to the final flow, using it to compute the shared secret, and comparing the result to the key chosen by the ideal functionality. Hofheinz et al. [HMS03] later showed that $\mathcal{F}_{ke}$ can in fact never be realized in the presence of adaptive adversaries. There are two simple ways to overcome this problem: modify the ideal functionality to allow the simulator to dictate the shared secret when a party is corrupted (the approach

taken by Dodis et al. [DKSW09], Hofheinz et al. [HMS03], and Unger and Goldberg [UG15]), or move part of the simulator into a protocol-specific *non-information oracle* (the approach taken by Canetti and Krawczyk [CK02b]).

A non-information oracle $\mathcal{N}$ is a probabilistic interactive Turing machine that interacts with another machine $\mathcal{M}$ and then produces local output. The "non-information" property requires that the local output of $\mathcal{N}$ is computationally indistinguishable from random from the perspective of $\mathcal{M}$, and independent of all messages exchanged between $\mathcal{N}$ and $\mathcal{M}$ [CK02b]. This construct can be used as part of an ideal functionality to make notions of key exchange realizable in the UC framework. Specifically, a properly designed non-information oracle can provide information to the simulator $\mathcal{S}$, allowing it to provide the "commitments" necessary to simulate a real protocol, while using secret internal state to provide a session key to the ideal functionality. The computational indistinguishability of the local output (which becomes the session key) from random guarantees that $\mathcal{S}$ (and thus $\mathcal{A}$) cannot compromise the shared secret key. Since the shared secret should not be hidden from the protocol participants, these internal secrets are exposed to $\mathcal{S}$ if it corrupts one of the participating ideal parties. Canetti and Krawczyk [CK02b] defined a relaxed ideal functionality, $\mathcal{F}_{wke}^{\mathcal{N}}$, parameterized by a non-information oracle $\mathcal{N}$, that is realized by a two-party DH key exchange. The security achieved by realizing this ideal functionality is equivalent to the older notion of SK-security [Can01].

Hofheinz et al. [HMS03] later noted that non-information oracles can be used to capture the common notion of contributiveness [HMS03]. This notion, which they refer to as *initiator resilience*, prevents the initiator of the key exchange from predetermining the value of the shared secret. As in a DH key exchange, the responder can still completely determine the value of the secret by selecting their contribution appropriately.

6.2.1.2 Deniability

Dodis et al. [DKSW09] noted that universal composability can be used to prove that a key exchange protocol is deniable. If a protocol realizes an appropriately chosen key exchange ideal functionality, then the resulting simulatability properties imply both offline and online deniability; the simulator $\mathcal{S}$ acts as the forger (in the offline case) or the misinformant (in the online case), and the distinguishing environment $\mathcal{Z}$ acts as the judge.

Dodis et al. [DKSW09] proposed the definition of a key exchange functionality with an "incriminating abort", which can be realized by efficient DAKEs that leak non-simulatable messages to active adversaries willing to cause session failures. This weakness is not a significant concern in practice, and it prevents the need to use unrealistically expensive cryptographic primitives. The simulator is permitted to ask the ideal functionality to abort the exchange, which prevents

the delivery of the shared secret to at least one party. After aborting the protocol, the simulator can obtain some non-simulatable information that betrays involvement in the protocol by one of the parties. The model parameterizes the functionality with a protocol-specific "incriminating procedure" called IncProc that generates this incriminating information from one of the long-term secret keys. The security proof sketches for RSDAKE and Spawn [UG15] adopted this approach.

6.2.1.3 Functionality Construction

Algorithm 6.2 depicts $\mathcal{F}^{+}_{post\text{-}keia}$, a new ideal functionality. This functionality incorporates the Hofheinz et al. [HMS03] model of contributiveness [HMS03], the Dodis et al. [DKSW09] model of deniability with incriminating abort [DKSW09], and the Unger and Goldberg [UG15] model of post-specified peers [UG15]. Consequently, the functionality is parameterized by both a non-information oracle $\mathcal{N}$ and an incrimination procedure IncProc.

The core of $\mathcal{F}^{+}_{post\text{-}keia}$ is the computation and delivery of a shared secret key. $\mathcal{F}^{+}_{post\text{-}keia}$ expects two parties to declare participation in the protocol with `initiate` and `establish` messages.[2] These parties are thereafter referred to as the initiator $\overline{I}$ and responder $\overline{R}$, respectively. Like Dodis et al. [DKSW09] [DKSW09, Fig. 3], all parties are assumed to have registered secret keys with the shared functionality before beginning a protocol session. $\mathcal{F}^{+}_{post\text{-}keia}$ ensures that the initiator is always defined first by delaying the processing of an `establish` message until a `initiate` message is seen. This simplifies simulator construction. Once both roles are defined, the simulator $\mathcal{S}$ is permitted to control delivery of the shared secret to $\overline{I}$ and $\overline{R}$. $\mathcal{S}$ sends an `ok` message to indicate that the shared key has become fixed. $\mathcal{S}$ can then individually choose to deliver the key to the ideal parties by sending `deliver` messages. Post-specified peers are modeled by allowing $\mathcal{S}$ to specify the identity of the remote party in `deliver` messages. $\mathcal{S}$ may only specify "incorrect" remote identities if it has corrupted the corresponding parties. As in Canetti and Krawczyk [CK02b], the *aux* parameter in the `initiate` and `establish` messages contains auxiliary routing information. Real protocols use this information to deliver messages to the other party even though their logical identity is not known at the start of the protocol. The ideal functionality simply ignores *aux*. The protocol-specific details of the message routing (e.g., local broadcasts, message pools, or central servers) are independent of the security analysis [CK02a]. When both parties receive the key, $\mathcal{F}^{+}_{post\text{-}keia}$ halts.

[2] Walfish [Wal08, Fig. 3.5], Dodis et al. [DKSW09, Fig. 3], and Unger and Goldberg [UG15, Alg. 1] all make a mistake in the registration messages by requiring the secret keys SK_I and SK_R as input. In UC/EUC/GUC, ideal functionalities are executed with *dummy parties* that simply pass inputs received from $\mathcal{Z}$ to the functionality. Since one purpose of using a shared functionality to model the PKI is to hide honest parties' secret keys from $\mathcal{Z}$, this prevents $\mathcal{Z}$ from starting the protocols. The correct solution, as noted in Section 6.1.2, is to give the ideal functionality access to the secret keys for the purpose of invoking IncProc. Walfish and Dodis et al. [DKSW09] both do this, but superfluously and erroneously also require SK_I and SK_R as inputs from the principal parties.

Algorithm 6.2 IDEAL FUNCTIONALITY MODELING DAKEZ'S BEHAVIOR. (Refs: 120 and 160)

Ideal Functionality | $\mathcal{F}^+_{post\text{-}keia}$

$\mathcal{F}^+_{post\text{-}keia}$ proceeds as follows, running on security parameter λ, in the $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$-hybrid model, with parties $\overline{P_1}, \ldots, \overline{P_n}$ and an adversary $\mathcal{S}$. The functionality is parameterized by a non-information oracle $\mathcal{N}$, and an incrimination procedure IncProc. When initializing, $\mathcal{F}^+_{post\text{-}keia}$ invokes $\mathcal{N}$ with fresh randomness.

on interaction with $\mathcal{N}$:
  Allow $\mathcal{S}$ to communicate with $\mathcal{N}$ by forwarding messages between them. If at any point $\bar{I}$ or $\bar{R}$ is corrupted or $\bar{R}$ is "aborted" while $\mathcal{N}$ has produced local output, send the complete state and output of $\mathcal{N}$ to $\mathcal{S}$.

on (`initiate`, $sid, \bar{I}, \Phi, aux$) **from** $\bar{P} \in \{\overline{P_1}, \ldots, \overline{P_n}\}$:
  if (an `initiate` message seen before) **return**
  if ($\bar{P} \neq \bar{I}$) **return**
  Record that $\bar{I}$ is the initiator
  Mark $\bar{I}$ as "active"
  Send (`initiate`, $sid, \bar{I}, \Phi$) to $\mathcal{S}$

on (`establish`, $sid, \bar{R}, \Phi$) **from** $\bar{P} \in \{\overline{P_1}, \ldots, \overline{P_n}\}$:
  if (an `establish` message seen before) **return**
  if ($\bar{P} \neq \bar{R}$) **return**
  if (initiator not recorded) {
    Resume processing once initiator is recorded
  }
  Record that $\bar{R}$ is the responder
  Mark $\bar{R}$ as "active"
  Send (`establish`, $sid, \bar{R}, \Phi$) to $\mathcal{S}$

on (`ok`, sid, k) **from** $\mathcal{S}$:
  if (a key tuple (sid, κ) has been recorded) **return**
  if (($\bar{I}$ not recorded) || ($\bar{R}$ not recorded)) **return**
  if (($\bar{I}$ is corrupt) **&&** ($\bar{R}$ is corrupt)) **return**
  if (($\bar{I}$ is uncorrupted) **&&** ($\bar{R}$ is uncorrupted)
  ↪ **&&** ($\bar{R}$ is "active")) { $\kappa \xleftarrow{\$} \{0,1\}^\lambda$ }
  else if (($\bar{I}$ is corrupt) **&&** ($\bar{R}$ is "active")) {
  ↪ Let κ denote the local output of $\mathcal{N}$ }
  else { $\kappa \leftarrow k$ }
  Record key tuple (sid, κ)

on (`deliver`, $sid, \bar{P}, \overline{P'}$) **from** $\mathcal{S}$:
  if (no key tuple (sid, κ) has been recorded) **return**
  if (a `set-key` message was already sent to $\bar{P}$) **return**
  if (IncProc was previously executed) **return**
  if (($\bar{P} \notin \{\bar{I}, \bar{R}\}$) || ($\bar{P}$ is not "active")) **return**
  if (($\{\bar{P}, \overline{P'}\} \neq \{\bar{I}, \bar{R}\}$) **&&** ($\overline{P'}$ is uncorrupted)) **return**
  Send (`set-key`, $sid, \overline{P'}, \kappa$) to $\bar{P}$
  if (two `set-key` messages have been sent) Halt

on (`abort`, sid) **from** $\mathcal{S}$:
  if ($\bar{I}$ is "active") Send delayed (`abort`, $sid, \bar{I}$) to $\bar{I}$
  if ($\bar{R}$ is "active") {
    Mark $\bar{R}$ as "aborted"
    Send delayed (`abort`, $sid, \bar{R}$) to $\bar{R}$
  }

on (`incriminate`, sid) **from** $\mathcal{S}$:
  if (IncProc was previously executed) **return**
  if (($\bar{R}$ is "aborted") **&&** ($\bar{I}$ is "active")
  ↪ **&&** ($\bar{R}$ is uncorrupted)) {
    Send (`retrievesecret`, $\bar{R}$) to $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$
    Retrieve SK_R from $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$
    Execute IncProc$(sid, \bar{I}, \bar{R}, PK_I, PK_R, SK_R)$
  }

Contributiveness is provided by placing restrictions on $\mathcal{S}$'s ability to set the value of the session key. If $\mathcal{S}$ does not corrupt either party before fixing the session key with an `ok` message, $\mathcal{F}^{+}_{\text{post-keia}}$ selects a key completely at random. However, if $\mathcal{S}$ corrupts only $\overline{I}$, its influence over the key is still restricted. In this case, $\mathcal{F}^{+}_{\text{post-keia}}$ draws the value of the session key from $\mathcal{N}$. Although $\mathcal{S}$ is granted unrestricted interaction with $\mathcal{N}$, the non-information property of $\mathcal{N}$ prevents $\mathcal{S}$ from learning or controlling the key. Only if $\mathcal{S}$ corrupts $\overline{R}$ before sending an `ok` message is it given complete control over the session key. In all cases, corruption of either party provides knowledge of the session key; $\mathcal{F}^{+}_{\text{post-keia}}$ transmits the internal state and output of $\mathcal{N}$ to $\mathcal{S}$ upon corruption of either participant.

To permit realization of the functionality, $\mathcal{F}^{+}_{\text{post-keia}}$ also provides an incriminating abort procedure through IncProc. If $\mathcal{S}$ sends an `abort` message to $\mathcal{F}^{+}_{\text{post-keia}}$ before the functionality halts, the key exchange can no longer fully complete. $\mathcal{F}^{+}_{\text{post-keia}}$ models this by internally labeling $\overline{I}$ and $\overline{R}$ (when defined) as "active" or "aborted". The session key cannot be delivered to an "aborted" party using a `deliver` message. For $\mathcal{F}^{+}_{\text{post-keia}}$, only $\overline{R}$ is guaranteed to be aborted; it is still possible for $\overline{I}$ to output a result. When aborting the protocol, $\mathcal{S}$ can choose to deliver notifications of the abort to active parties independently. If both parties have been defined, $\overline{R}$ remains uncorrupted, and $\mathcal{S}$ has aborted the protocol, $\mathcal{S}$ can send an `incriminate` message to trigger invocation of `IncProc`, allowing it to receive protocol-specific incriminating messages.

One subtle interaction between the features is that $\mathcal{S}$ must be allowed to derive the session key when the protocol is aborted. This case models the real-world situation in which the adversary has altered a message flow to incorporate adversarially controlled ephemeral state. While an authenticated key exchange must detect this alteration and prevent completion of the protocol (modeled in $\mathcal{F}^{+}_{\text{post-keia}}$ by the abort procedure), the attack may allow the adversary to derive the shared secret after it has become fixed, but before verification occurs.

6.2.2 DAKEZ in the GUC Framework

DAKEZ was defined in Section 5.4. However, to prove the security of DAKEZ in the GUC framework, it is necessary to define the DAKEZ protocol in terms of the $\mathcal{F}^{+}_{\text{post-keia}}$ interface. An interactive Turing machine that completes a DAKEZ key exchange in a way that is indistinguishable from dummy parties forwarding the same inputs to $\mathcal{F}^{+}_{\text{post-keia}}$ must be defined. The protocol steps are the same as in Figure 5.2, but the interface must be changed. Algorithm 6.3 contains the adapted protocol. The protocol is implicitly parameterized with the group $\mathbb{G}$, q, and g.

After receiving its input, the DAKEZ protocol in Algorithm 6.3 determines whether it is playing the role of the initiator or the responder. The notation "**on** (m) **to** $\mathcal{P}$" means that the

Algorithm 6.3 DAKEZ IMPLEMENTED IN THE GUC FRAMEWORK. (Refs: 122[a b], 124[a b], and 140)

Real Protocol | DAKEZ

on activation with input $(\texttt{initiate}, sid, \mathcal{I}, \Phi, aux)$:
  Record that we are the initiator, $\mathcal{I}$
  Retrieve PK_I and SK_I from shared functionality
  Record PK_I, SK_I, sid, and Φ
  Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$
  Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{R}, \Phi)$:
  Record that we are the responder, $\mathcal{R}$
  Retrieve PK_R and SK_R from shared functionality
  Record PK_R, SK_R, sid, and Φ
  Set state to $\texttt{await-}\psi_1$

on $(\mathcal{P} \| g^p \| PQ_P)$ **to** $\mathcal{R}$ **in state** $\texttt{await-}\psi_1$:
  Record $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_P)$
  Record $\mathcal{P}$, g^p, PQ_P, g^r, and Q_R
  Retrieve PK_P from shared functionality
  Let $t =$ "0"$\|\mathcal{P}\|\mathcal{R}\|g^p\|g^r\|PQ_P\|Q_R\|\Phi$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_P, PK_R, g^p\}, t)$
  Record $k = \text{KDF}((g^p)^r \| Q_k)$
  Erase r and Q_k
  Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| \sigma$ to $\mathcal{P}$
  Set state to $\texttt{await-}\psi_2$

on $(\mathcal{P} \| g^p \| Q_P \| \sigma)$ **to** $\mathcal{I}$:
  Retrieve PK_P from shared functionality
  Let $t_1 =$ "0"$\|\mathcal{I}\|\mathcal{P}\|g^i\|g^p\|PQ_I\|Q_P\|\Phi$
  if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t_1)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Compute $Q_k = \text{QRDecaps}(SQ_I, Q_P)$
  Compute $k = \text{KDF}((g^p)^i \| Q_k)$
  Erase i, SQ_I, and Q_k
  Let $t_2 =$ "1"$\|\mathcal{I}\|\mathcal{P}\|g^i\|g^p\|PQ_I\|Q_P\|\Phi$
  Compute $\psi_3 = \text{RSig}(PK_I, SK_I, \{PK_I, PK_P, g^p\}, t_2)$
  Send ψ_3 to $\mathcal{P}$
  Locally output $(\texttt{set-key}, sid, \mathcal{P}, k)$ and halt

on (σ) **to** $\mathcal{R}$ **in state** $\texttt{await-}\psi_2$:
  Let $t =$ "1"$\|\mathcal{P}\|\mathcal{R}\|g^p\|g^r\|PQ_P\|Q_R\|\Phi$
  if $(\neg(\text{RVrf}(\{PK_P, PK_R, g^r\}, \sigma, t_1)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  }
  Locally output $(\texttt{set-key}, sid, \mathcal{P}, k)$ and halt

on unknown or invalid message:
  Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
  Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 6.4 INCRIMINATION PROCEDURE FOR DAKEZ. (Ref: 124)

Subroutine | $\text{IncProc}_{\text{DAKEZ}}(sid, \bar{I}, \bar{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \bar{I}, \bar{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
  Let $t = \text{“0”}\|\text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
  Compute $\psi = \text{“}R\text{”}\|g^r\|Q_R\|\sigma$
  Send $(\texttt{inc}, sid, \bar{I}, \bar{R}, \psi, g^r, r, Q_R, Q_k)$ to $\mathcal{S}$

given function is executed when a message of the form m is received, and the party is playing the role of $\mathcal{P}$. When evaluating the form of a message, group elements are checked to ensure that they are in $\mathbb{G}$ and are not the identity element. All variables shown in Algorithm 6.3 are scoped to their containing function unless they are explicitly persisted using the “Record” statement (e.g., the variable t in one function is not the same as the t in another function).

The shared functionality for DAKEZ, $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$, is defined to be $\bar{\mathcal{G}}^{\text{DAKEZ},2,\mathbb{G},q,g}_{krkro}$. All calls to the key derivation function $\text{KDF}(x)$ are modeled by the first random oracle; a message $(\texttt{ro}, 1, x)$ is sent to $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$, which then replies with the result. In a similar manner, the random oracle needed to model the hash function within the RSig scheme (thereby making its security proofs applicable) is provided by the second random oracle in $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$.

6.2.3 Proof Strategy

This section describes a strategy for proving the security of DAKEZ by showing that it GUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$. The three message flows of DAKEZ, as shown in Algorithm 6.3, are denoted as ψ_1, ψ_2, and ψ_3. Since $\mathcal{F}^{+}_{post\text{-}keia}$ is parameterized by additional procedures, an incrimination procedure and non-information oracle for DAKEZ is needed. Algorithm 6.4 depicts the DAKEZ incrimination procedure, which simply computes ψ_2 as an honest responder would. In practice, the presence of IncProc allows an adversary to prove that a party is willing to respond to a key exchange request from an entity with a particular claimed (but unauthenticated) identity. For most applications, this has no real-world impact on the security or privacy of the protocol.

Algorithm 6.5 depicts the non-information oracle for DAKEZ. The general construction of this oracle follows the approach of Hofheinz et al. [HMS03] for DH-based protocols. $\mathcal{N}_{QRDH}$ internally generates both ephemeral keys and sends the public parts to $\mathcal{M}$. $\mathcal{M}$ can then accept or reject the proposed keys by sending a `complete` message. If the keys are accepted (by sending a `complete`

message with ok = TRUE), $\mathcal{N}_{QRDH}$ completes the exchange and locally outputs the shared secret. If $\mathcal{M}$ rejects the keys, $\mathcal{N}_{QRDH}$ discards them and accepts one half of the ephemeral key exchange from $\mathcal{M}$ (in the α and β parameters). The local output of $\mathcal{N}_{QRDH}$ in this case is the shared secret of an exchange with $\mathcal{M}$. This option is necessary in the event that the adversary corrupts the initiator in the exchange; $\mathcal{N}$ grants the simulator the ability to complete the exchange on behalf of the remaining honest simulated party. $\mathcal{N}$ also provides a facility to generate RSig messages using either ephemeral key held by $\mathcal{N}_{QRDH}$.

While this section does not formally prove it, $\mathcal{N}_{QRDH}$ is clearly a non-information oracle for appropriately chosen cryptographic groups. The only information revealed to $\mathcal{M}$ are public DH contributions and public values produced by QRGen and QREncaps. The only input from $\mathcal{M}$ is α and β when the initial exchange has been rejected. In this case, $\mathcal{M}$ never receives enough information to complete the exchange. The release of RSig proofs in response to `prove` messages releases no information other than the possession of a key in the set S by design. Consequently, distinguishing the output of $\mathcal{N}_{QRDH}$ from random would require $\mathcal{M}$ to break the CDH assumption in $\mathbb{G}$ (in order to determine the correct input to the random oracle) or the zero-knowledge property of the SoK in RSig.

The security theorem for DAKEZ is as follows:

Theorem 3 | DAKEZ is secure

If the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then DAKEZ GUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$-hybrid model with adaptive security for $\text{IncProc}_{\text{DAKEZ}}$ and non-information oracle $\mathcal{N}_{QRDH}$.
(Refs: 125, 127[abc], 128, 129, 153, and 502)

The proof uses the erasure model defined by Dodis et al. [DKSW09], which allows participants in the protocol to erase private state information, preventing it from being leaked in the event of a subsequent corruption. Since the capability to securely erase RAM contents is generally accepted in practice, this weakening of the model does not admit any actual attacks. The construction of the simulator for the proof of Theorem 3 can be used as guidance for practitioners seeking to implement real-world key exchange forgery tools (as a means to improve the plausible deniability of higher-level protocols). In general, the simulator simulates both parties involved in the key exchange honestly. To produce the RSig proofs, the simulator uses the ephemeral keys of the remote party in the case when the adversary has not corrupted either participant, or the compromised long-term keys if a participant has been corrupted. The simulator uses the non-information oracle to simulate the calculation of the shared secret key, and uses its access to the non-information oracle's internal state to construct the simulated memory contents of

Algorithm 6.5 NON-INFORMATION ORACLE FOR DAKEZ. (Refs: 124 and 149)

Subroutine | $\mathcal{N}_{QRDH}$

on $(\texttt{setup}, \mathbb{G}, q, g)$ **from** $\mathcal{M}$:
  if (a `setup` message was already received) **return**
  Read $(\mathbb{G}, q, g)$ as group $\mathbb{G}$, prime order q, generator g
  Generate $i \xleftarrow{\$} \mathbb{Z}_q$ and $r \xleftarrow{\$} \mathbb{Z}_q$
  Generate $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$
  Generate $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
  Record $\mathbb{G}$, q, g, i, r, SQ_I, and Q_k
  Send $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ to $\mathcal{M}$

on $(\texttt{complete}, ok, \alpha, \beta)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (already output a key) **return**
  if (ok is TRUE) {
    Compute $k = \text{KDF}((g^i)^r \parallel Q_k)$
  } **else** {
    if $((\alpha \notin \mathbb{G}) \,||\, (\alpha$ is identity element$))$ **return**
    if (β not generated by QRGen) **return**
    Generate new $r \xleftarrow{\$} \mathbb{Z}_q$
    Generate new $(Q_R, Q_k) \leftarrow \text{QREncaps}(\beta)$
    Compute $k = \text{KDF}(\alpha^r \parallel Q_k)$
  }
  Locally output k

on $(\texttt{prove}, p, S, m)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  if $(p \notin \{1, 2\})$ **return**
  if $(p = 1)$ { Let $x \leftarrow i$ } **else** { Let $x \leftarrow r$ }
  if $(g^x \notin S)$ **return**
  Compute $\sigma = \text{RSig}(g^x, x, S, m)$
  Send $(\texttt{proof}, \sigma)$ to $\mathcal{M}$

corrupted parties. The simulator only uses IncProc if the first message, ψ_1, is altered by the adversary.

6.2.4 Relationship to Security Properties

Now that the ideal functionality that represents the features of DAKEZ and stated the associated security theorem have been defined, the security properties in Section 5.2.2 can be rigorously defined in terms of the GUC framework. Each property either follows from the definition of $\mathcal{F}^{+}_{post\text{-}keia}$, or from the proof of Theorem 3:

1. **Universally composable AKE**: $\mathcal{F}^{+}_{post\text{-}keia}$ provides mutual authentication, key secrecy, and key freshness. Mutual authentication ensures that $\mathcal{S}$ (and thus $\mathcal{A}$) cannot cause $\overline{I}$ (resp. $\overline{R}$) to output an uncorrupted partner identifier other than $\overline{R}$ (resp. $\overline{I}$). Key secrecy and freshness ensure that if $\overline{I}$ or $\overline{R}$ is uncorrupted and outputs a key κ and a partner identifier $\overline{P}$, and $\overline{P}$ is uncorrupted, then $\mathcal{Z}$ cannot distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^{\lambda}$.

2. **Offline deniability**: It is possible to construct a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between $\mathcal{S}$ interacting with $\mathcal{F}^{+}_{post\text{-}keia}$ and $\mathcal{A}$ interacting with DAKEZ after the following sequence of events: $\mathcal{Z}$ selects a party $\overline{P} \in \{\overline{I}, \overline{R}\}$, allows the key exchange to complete unmodified, corrupts both $\overline{I}$ and $\overline{R}$, and asks $\mathcal{S}$ to reveal $\overline{P}$'s ephemeral keys. Such a simulator can be constructed from the proof of Theorem 3 by simulating the "honest" case and then exposing the contents of the non-information oracle to $\mathcal{Z}$.

3. **Online deniability**: $\mathcal{Z}$ cannot distinguish between $\mathcal{S}$ interacting with $\mathcal{F}^{+}_{post\text{-}keia}$ and $\mathcal{A}$ interacting with DAKEZ, given the restrictions in Theorem 3.

4. **Contributiveness / Initiator-resilience**: When $\overline{R}$ is uncorrupted, any key output by $\overline{R}$ is computationally independent of any values chosen by $\mathcal{Z}$. This property follows from the design of $\mathcal{F}^{+}_{post\text{-}keia}$, which either outputs a bit string chosen uniformly at random or the local output of the non-information oracle when $\overline{R}$ is uncorrupted.

5. **Forward secrecy**: In the strong form [BPR00]: if a party $\overline{P}$ outputs a key κ and a partner identifier $\overline{P'}$, then $\mathcal{Z}$ can never distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^{\lambda}$ unless $\overline{P}$ or $\overline{P'}$ was corrupted before the corresponding session completed. In the weak form of the property, this is only true if $\mathcal{S}$ also did not abort the session (and thus $\mathcal{A}$ did not modify any messages).

6. **Post-specified peer**: The `initiate` and `establish` messages do not identify the intended communication partner.

6.2.4.1 Quantum Transitional Security

Note that Theorem 3 only refers to classical security (i.e., it ignores quantum adversaries). This is also true of the other theorems presented in this chapter. Quantum transitionally secure key exchanges cannot realize authenticated key exchange functionalities like $\mathcal{F}^{+}_{post\text{-}keia}$ against quantum adversaries because, by definition, they fail to provide the necessary authentication properties in the Q^{c}Q setting (see Section 5.2.3). However, a more traditional analysis makes it clear that DAKEZ is secure in the C^{c}Q setting (i.e., it provides quantum transitional security). Because the KDF is modeled by a random oracle, any passive adversary that can derive the session key using only long-term secret keys and a transcript of the exchange must be able to derive the KDF input and send it to $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ within an `ro` message. This KDF input includes Q_k, the shared secret derived from the QREncaps and QRDecaps functions. Since the adversary does not have access to ephemeral state ($\mathcal{I}$ and $\mathcal{R}$ already erased SQ_I and Q_k from their memory), the adversary must be able to derive Q_k using only PQ_I and Q_R (and the unrelated secrets I and R). Therefore, this passive adversary can break the key secrecy property of the quantum-resistant KEM, which is assumed to be impossible due to the IND-CPA security of the KEM. Moreover, it is not possible for an active adversary to modify the KEM transmissions because PQ_I and Q_R are authenticated by the ring signatures. In the C^{c}Q setting, the adversary does not have access to quantum computation when the protocol session occurs, so it is not able to break the classical unforgeability of RSig. Consequently, quantum IND-CPA security is sufficient for the quantum-resistant KEM—there is no need for quantum IND-CCA2 security. A similar argument holds for the other DAKEs discussed in this chapter. Combining an IND-CPA quantum-resistant KEM with a classically secure authentication mechanism to achieve security in the C^{c}Q setting is similar to the approach used by Bindel et al. [BBF+19, §4.3] to add quantum transitional security to a SIGMA key exchange protocol.

As discussed in Section 5.3.4, some quantum-resistant KEMs, such as New Hope [ADPS16], are only δ-correct—the QRDecaps function can fail for valid inputs with probability δ. The ideal functionalities in this chapter assume that this is not possible and that the quantum-resistant KEM is completely correct. If δ is negligible (e.g., $2^{-\lambda}$ for security parameter λ), as it is for all of the finalists and alternate candidates in the third round of the NIST standardization effort [AAA+20], then this is not a problem. In particular, it is acceptable for the ideal functionalities to output matching session keys for the DAKE participants in an honest case where QRDecaps would return $\perp$ in reality, because these situations are overwhelmingly improbable and can be safely ignored.

6.2.5 Proof of Theorem 3

(Sketch) To show that DAKEZ GUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$, it suffices to show that DAKEZ EUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$. DAKEZ EUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$ if and only if, for any PPT adversary $\mathcal{A}$ attacking DAKEZ, there exists a PPT adversary $\mathcal{S}$ attacking $\mathcal{F}^{+}_{post\text{-}keia}$ such that any $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$-externally constrained environment $\mathcal{Z}$ cannot distinguish between the real and simulated conditions.

Like most proofs in UC-based models, this proof will construct a simulator $\mathcal{S}$ that executes $\mathcal{A}$ internally, simulating the real protocol flows that $\mathcal{A}$ expects based on conditions in the ideal environment. For any ideal party $\overline{P}$, $\mathcal{S}$ simulates a party $P^{(s)}$ for $\mathcal{A}$. All parties know the shared protocol parameters used to instantiate $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$: a group $\mathbb{G}$ of prime order q with generator g. To achieve the required indistinguishability property, two things must be shown: $\mathcal{Z}$ can derive no useful information from sessions other than the one under consideration, and $\mathcal{Z}$ cannot distinguish between the challenge protocols in the context of the current session. To guarantee the latter condition, it must be shown that, irrespective of the actions performed by $\mathcal{A}$ under the instruction of $\mathcal{Z}$, the outputs of the main parties of $\mathcal{F}^{+}_{post\text{-}keia}$ are equal to those of DAKEZ, corrupted parties provide memory consistent with all other observations, and the protocol flows within the joint view of $\mathcal{A}$ and $\mathcal{Z}$ are consistent with the outputs of the main parties.

Section 6.2.6 describes the simulator construction, and Section 6.2.7 presents the indistinguishability proof.

6.2.6 Simulator Construction

6.2.6.1 Communications between $\mathcal{A}$ and $\mathcal{Z}$

Any data sent to $\mathcal{S}$ from $\mathcal{Z}$ are copied to the input of $\mathcal{A}$. Likewise, any output from $\mathcal{A}$ is sent to $\mathcal{Z}$ by $\mathcal{S}$.

6.2.6.2 General reactions to actions by $\mathcal{A}$

If $\mathcal{A}$ sends any messages within the simulated environment that are unrelated to DAKEZ, they are ignored (as they would be in a real network environment). If $\mathcal{A}$ delays delivery of a message flow, $\mathcal{S}$ simply waits for the flow to be delivered before continuing. This leaves $\mathcal{A}$ with few possible actions of consequence: it can alter any of the message flows it perceives (this is equivalent to delaying a message and sending a different one in its place), and it can corrupt simulated parties. The model allows $\mathcal{A}$ to corrupt parties before the protocol begins, after ψ_1 has been sent, after

ψ_2 has been sent, or after ψ_3 has been sent (i.e., corruptions can be fully adaptive). When $\mathcal{A}$ corrupts a simulated party, $\mathcal{S}$ corrupts the corresponding ideal party in order to construct the expected state history. If $\mathcal{A}$ causes a corrupted simulated party to output a message, $\mathcal{S}$ causes the corresponding ideal party to output the same message.

6.2.6.3 Initialization

When $\mathcal{S}$ first initializes, it sends a $(\texttt{setup}, \mathbb{G}, q, g)$ message to $\mathcal{N}$ through $\mathcal{F}^{+}_{\textit{post-keia}}$, and waits to receive a $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ message in response. The group details sent by $\mathcal{S}$ correspond to the protocol instantiation attacked by $\mathcal{A}$. $\mathcal{S}$ makes a note of the ephemeral keys in the `exchange` message for later use in the simulation.

6.2.6.4 Receipt of `initiate` message from $\mathcal{F}^{+}_{\textit{post-keia}}$

When $\mathcal{S}$ receives $(\texttt{initiate}, \textit{sid}, \overline{I}, \Phi_I)$ from $\mathcal{F}^{+}_{\textit{post-keia}}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ computes $\psi_1 =$ "I" $\|\ g^i \|\ PQ_I$ using the g^i and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ also records the value Φ_I for later reference.

6.2.6.5 Receipt of `establish` message from $\mathcal{F}^{+}_{\textit{post-keia}}$

When $\mathcal{S}$ receives an establishment message $(\texttt{establish}, \textit{sid}, \overline{R}, \Phi_R)$ from $\mathcal{F}^{+}_{\textit{post-keia}}$, it checks to see the circumstances of the simulated ψ_1 message transmission. Since $\mathcal{F}^{+}_{\textit{post-keia}}$ only sends an `establish` message after it has already sent an `initiate` message, ψ_1 is guaranteed to have been sent and received in the simulated environment (either by $\mathcal{S}$ in response to an `initiate` message or by $\mathcal{A}$ from a corrupted party).

$\mathcal{S}$ parses ψ_1 to recover g^i and PQ_I. If ψ_1 is not of the correct format, or it fails to validate (e.g., if "I" is not a valid identity), then $\mathcal{S}$ sends $(\texttt{abort}, \textit{sid})$ to $\mathcal{F}^{+}_{\textit{post-keia}}$ and delivers the resulting `abort` message to $\overline{R}$ immediately. $\mathcal{S}$ withholds the `abort` message to $\overline{I}$.

$\mathcal{S}$ records Φ_R for later reference.

If ψ_1 is valid, then $\mathcal{S}$ constructs a message ψ_2 from $R^{(s)}$ in response to ψ_1. The mechanism for constructing ψ_2 depends on how ψ_1 was generated:

- If $\mathcal{S}$ previously created ψ_1 in response to an `initiate` message, then $\mathcal{S}$ uses the non-information oracle $\mathcal{N}$ to construct ψ_2. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending $(\texttt{prove}, 1, \{g^I, g^R, g^i\}, t)$, where t = "0"‖"I"‖"R"‖g^i‖g^r‖PQ_I‖Q_R‖Φ_R, to $\mathcal{N}$ and waiting for a message $(\texttt{proof}, \sigma_R)$ in response. $\mathcal{S}$ then constructs ψ_2 = "R" ‖ g^r ‖ Q_R ‖ σ_R.

- If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ uses its access to corrupt $\overline{I}$ to retrieve $SK_I = I$ from $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ using a `rectrievesecret` message. $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I)$. Since $\overline{I}$ is corrupted, $\mathcal{F}^+_{post\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ sends $(\texttt{ok}, sid, 0)$ to $\mathcal{F}^+_{post\text{-}keia}$, causing it to record the output from $\mathcal{N}$ as the shared key. $\mathcal{S}$ uses the newly generated values g^r and Q_R to construct ψ_2 = "R" ‖ g^r ‖ Q_R ‖ σ_R. $\mathcal{S}$ calculates σ_R as in the previous case, except that it uses the long-term keypair (g^I, I) to produce the proof.

- If $\mathcal{S}$ previously created a message ψ_1' but $\psi_1 \neq \psi_1'$, then $\mathcal{A}$ has altered the message in transit. $\mathcal{S}$ constructs ψ_2 through the use of IncProc. $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{post\text{-}keia}$, but withholds delivery of the resulting `abort` messages to $\overline{I}$ and $\overline{R}$. It then sends $(\texttt{incriminate}, sid)$ to $\mathcal{F}^+_{post\text{-}keia}$, causing an instance of IncProc to be invoked. Using the values parsed from ψ_1, $\mathcal{S}$ sends $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{"}I\text{"}, \text{"}R\text{"}, \Phi_R, g^i, PQ_I)$ to the instance of IncProc and receives $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi_2, g^r, r, Q_R, Q_k)$ in response.

$\mathcal{S}$ then sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$.

6.2.6.6 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract "P", g^p, Q_P, and the proof σ_P. If ψ_2 is not of the correct form, or if σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{post\text{-}keia}$ and delivers the resulting `abort` message to $\overline{I}$ immediately, while withholding any `abort` message sent to $\overline{P}$.

If $I^{(s)}$ has previously broadcast a message ψ_1 and ψ_2 is valid, then $\mathcal{S}$ constructs message ψ_3 to send from $I^{(s)}$ to $P^{(s)}$ and outputs a shared key. The private key that $\mathcal{S}$ uses to construct the proof ψ_3, and the shared key that it outputs, depends on the state of the simulation:

- If $\mathcal{A}$ has previously corrupted $P^{(s)}$, then $\mathcal{S}$ must have previously corrupted $\overline{P}$ (since $\mathcal{S}$ corrupts ideal parties corresponding to simulated parties corrupted by $\mathcal{A}$). In this case, $\mathcal{S}$ retrieves $SK_P = P$ from $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ using a `retrievesecret` message and uses this key to compute $\psi_3 = \text{RSig}(g^P, P, \{g^I, g^P, g^p\}, t)$, where the tag t is given by t = "1"‖"I"‖"P"‖g^i‖g^p‖PQ_I‖Q_P‖Φ_I.

$\bar{I}$ is expected to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. If $\mathcal{S}$ previously simulated a ψ_2 message from a party $R^{(s)}$ and $R^{(s)} \neq P^{(s)}$, then $\mathcal{S}$ issues a $(\texttt{abort}, sid)$ message to $\mathcal{F}^{+}_{post\text{-}keia}$ and withholds the resulting `abort` messages. In any case, $\mathcal{F}^{+}_{post\text{-}keia}$ receives the internal state of $\mathcal{N}$ from $\mathcal{S}$, allowing it to acquire the ephemeral keys i and SQ_I used to generate ψ_1 (this occurs because either $\overline{R}$ has been aborted, or $\overline{R}$ is corrupt). $\mathcal{S}$ computes $k = \text{KDF}((g^p)^i \parallel \text{QRDecaps}(SQ_I, Q_P))$ and sends $(\texttt{ok}, sid, k)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing it to record key tuple (sid, k).

- Otherwise, $P^{(s)}$ is uncorrupted. Because RSig is unforgeable with respect to insider corruption, the only way for σ_P to be valid in this situation is if the sender of the message knows the long-term private key of one of the parties, or the ephemeral key of $I^{(s)}$. Additionally, σ_P must have been computed during this session because its validity depends on ψ_1. Due to the hardness of the discrete log problem in the group (guaranteeing the secrecy of the keys) and the uniqueness of ψ_1, this is only possible if $P^{(s)} = R^{(s)}$ and ψ_2 was previously generated by $\mathcal{S}$. $\mathcal{S}$ forges the proof ψ_3 using $R^{(s)}$'s ephemeral key held by the non-information oracle $\mathcal{N}$. $\mathcal{S}$ sends $(\texttt{prove}, 2, \{g^I, g^R, g^r\}, t)$, where the tag t is given by $t = \text{“1”}\|\text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi_I$, to $\mathcal{N}$ through $\mathcal{F}^{+}_{post\text{-}keia}$ and receives $(\texttt{proof}, \psi_3)$ in response. $\mathcal{S}$ sends $(\texttt{ok}, sid, 0)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing it to record a random session key.

$\mathcal{S}$ sends $(\texttt{deliver}, sid, \bar{I}, \overline{P})$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing $\bar{I}$ to emit the proper shared secret.

6.2.6.7 Receipt of ψ_3 by uncorrupted $R^{(s)}$

When uncorrupted $R^{(s)}$ receives message ψ_3 from $I^{(s)}$, $\mathcal{S}$ first checks to ensure that $R^{(s)}$ has previously received a message ψ_1 from $I^{(s)}$ and that it sent a response ψ_2. If either of these conditions do not hold, then the message is ignored. $\mathcal{S}$ then verifies the proof in ψ_3.

If the proof is invalid, does not match Φ_R or the ψ_2 message previously sent by $R^{(s)}$, or fails to verify, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$ and delivers the resulting `abort` message to $\overline{R}$ immediately. $\mathcal{S}$ withholds the `abort` message sent to $\bar{I}$.

If the proof is valid, then $\mathcal{S}$ also causes $\overline{R}$ to output a key. Since RSig is unforgeable with respect to insider corruption, it is only possible to reach this state if $I^{(s)}$ is corrupt or if the exchange has completed honestly; in all situations, $\mathcal{S}$ has already caused $\mathcal{F}^{+}_{post\text{-}keia}$ to record a shared key. $\mathcal{S}$ sends $(\texttt{deliver}, sid, \overline{R}, \bar{I})$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing $\overline{R}$ to emit the proper shared secret.

6.2.6.8 Transmission of ψ_1 by corrupted $I^{(s)}$

When $\mathcal{S}$ has not yet received an `initiate` message from $\mathcal{F}^{+}_{post\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $I^{(s)}$ to issue message ψ_1, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\bar{I}$ to send $(\texttt{initiate}, sid, \bar{I}, \perp, \perp)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, but ignores the resulting `initiate` message sent by $\mathcal{F}^{+}_{post\text{-}keia}$.

6.2.6.9 Transmission of ψ_2 by corrupted $R^{(s)}$

When $\mathcal{S}$ has not yet received an `establish` message from $\mathcal{F}^{+}_{post\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $R^{(s)}$ to issue message ψ_2, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{R}$ to send $(\texttt{establish}, sid, \overline{R}, \perp)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, but ignores the resulting `establish` message sent by $\mathcal{F}^{+}_{post\text{-}keia}$.

6.2.6.10 Constructing state for corrupted parties

When $\mathcal{A}$ corrupts a party in the simulated environment, $\mathcal{S}$ corrupts the corresponding party in the ideal environment. If $\mathcal{A}$ causes corrupted parties to output values, $\mathcal{S}$ outputs these values from the corresponding ideal parties. In addition, $\mathcal{S}$ must provide $\mathcal{A}$ with a simulated historical state for corrupted parties.

If $\mathcal{A}$ corrupts the party known as $I^{(s)}$ after an `initiate` message has been received, but before $I^{(s)}$ has received ψ_2, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to provide the random coins i, and SQ_I used to construct ψ_1. If $I^{(s)}$ already received ψ_2, then $\mathcal{S}$ uses its corruption of $\bar{I}$ to provide the session key k that $\bar{I}$ already output.

If $\mathcal{A}$ corrupts the party known as $R^{(s)}$ after it has already received ψ_1, then it must provide the session key k expected to be stored in $R^{(s)}$'s memory. If $R^{(s)}$ has already received ψ_3, then $\overline{R}$ has already output session key k, and so $\mathcal{S}$ can directly provide this value. If $R^{(s)}$ has not yet received ψ_3, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to obtain k (in the event that ψ_1 was sent by a corrupt $I^{(s)}$), or the random coins r and Q_k that, together with ψ_1, can be used to compute k.

6.2.7 Proof of Indistinguishability

The next task is to prove that $\mathcal{S}$ acting on $\mathcal{F}^{+}_{post\text{-}keia}$ is indistinguishable from $\mathcal{A}$ acting on DAKEZ. To do this, all possible behaviors of $\mathcal{A}$ are divided into several cases. For each case, it must be shown that the protocol flows generated by $\mathcal{S}$ are indistinguishable from those generated

by DAKEZ, outputs from $\mathcal{F}^{+}_{post\text{-}keia}$ are indistinguishable from those from DAKEZ, and that the simulated memory states of corrupted parties are indistinguishable from those of real parties.

6.2.7.1 The honest case

This case occurs when $\mathcal{A}$ does not alter any messages or corrupt $I^{(s)}$ or $R^{(s)}$ until after the session concludes.

All three messages are generated by the combination of $\mathcal{S}$ and $\mathcal{N}$ honestly (i.e., exactly how they would be generated by the parties in a real DAKEZ session), with the exception of the proofs. The proofs are not signed by the long-term secret keys of the parties, as in a real interaction. Instead, they are produced by $\mathcal{N}$ using the ephemeral key of the opposite party. However, because RSig is anonymous against full key exposure, $\mathcal{Z}$ cannot distinguish the proofs produced by $\mathcal{S}$ from those produced in a real interaction, even when it corrupts the long-term keys of $I^{(s)}$ and $R^{(s)}$ after the session.

If the shared session state provided to the parties by $\mathcal{Z}$ differs (i.e., $\Phi_I \neq \Phi_R$), then $I^{(s)}$ will immediately abort when ψ_2 is delivered. This output is the same as in the real protocol because the SoK in ψ_2 will be bound to the wrong message.

When $\Phi_I = \Phi_R$, the output from $\overline{I}$ and $\overline{R}$ in the ideal environment includes the correct identity of the conversation partner, as well as a shared secret k randomly generated by $\mathcal{F}^{+}_{post\text{-}keia}$. These are the expected party identities from the real interaction, so the only possible way for $\mathcal{Z}$ to distinguish between real and simulated outputs is by examining k. Since i, r, SQ_I, and Q_k are erased by real parties before they return output, $\mathcal{A}$ cannot access these values, even when corrupting $I^{(s)}$ and $R^{(s)}$ after the protocol concludes. Therefore, any ability to distinguish between challenge protocols based on the choice of k would mean that $\mathcal{Z}$ could distinguish between k and $\mathrm{KDF}(g^{ir} \parallel Q_k)$. Since KDF is modeled by the random oracle in $\bar{\mathcal{G}}^{\mathrm{DAKEZ}}_{krkro}$, this is only possible if $\mathcal{Z}$ can break the CDH assumption in $\mathbb{G}$ (and the one-way security of the quantum-resistant KEM, if one is used) and send the message $(\mathtt{ro}, 1, g^{ir} \parallel Q_k)$ to $\bar{\mathcal{G}}^{\mathrm{DAKEZ}}_{krkro}$, which is assumed to be impossible.

In the event that a quantum-resistant KEM is used (i.e., the QRGen, QREncaps, and QRDecaps functions are not the "no-op" $\mathrm{QRGen}^{\varnothing}$, $\mathrm{QREncaps}^{\varnothing}$, and $\mathrm{QRDecaps}^{\varnothing}$ implementations defined in Section 5.3.4) and QRDecaps is δ-correct for some $0 < \delta \leq 2^{-\lambda}$ where λ is the security parameter, then there is a possibility that QRDecaps may return $\perp$ in the honest case. In this situation, $\mathcal{I}$ would fail to derive the same secret k as $\mathcal{R}$ when executing the real protocol, whereas $\mathcal{F}^{+}_{post\text{-}keia}$ would output a randomly generated shared secret for $\overline{I}$ and $\overline{R}$, theoretically allowing $\mathcal{Z}$ to distinguish between the settings. However, this event occurs with negligible probability δ, so it cannot be used by $\mathcal{Z}$ to gain advantage in the distinguishability game.

6.2.7.2 Alteration of ψ_1

This situation occurs when ψ_1 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ are corrupted when ψ_1 is delivered.

When ψ_1 is altered, $\mathcal{S}$ generates ψ_2 from $R^{(s)}$ using IncProc. The definition of IncProc involves honestly generating ψ_2 using the long-term secret key of $\overline{R}$, so this flow is indistinguishable from a real message. Likewise, the memory state of $R^{(s)}$ is indistinguishable from the real situation because IncProc provides $\mathcal{S}$ with the random coins used to generate the ephemeral keys in ψ_2 and the session key k.

$\mathcal{S}$ causes the protocol to abort, but does not deliver abort messages to either party. If $\mathcal{A}$ allows ψ_2 to be delivered to $I^{(s)}$, then $I^{(s)}$ will abort. This matches the output of real interactions because $I^{(s)}$ expects ψ_2 to include a proof incorporating the true g^i and PQ_I values sent by $I^{(s)}$ in ψ_1. The only way for the simulated and real situations to differ is if $\mathcal{A}$ somehow alters ψ_2 so that it is a valid response. Since RSig is unforgeable with respect to insider corruption, this is not possible.

6.2.7.3 Alteration of ψ_2

This situation occurs when ψ_2 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ are corrupted when ψ_2 is delivered. $\mathcal{S}$ causes $I^{(s)}$ to immediately abort when it receives an altered ψ_2. As mentioned previously, $I^{(s)}$ will always abort because the proof in the altered ψ_2 message cannot be correct due to RSig being unforgeable with respect to insider corruption.

6.2.7.4 Alteration of ψ_3

This situation occurs when ψ_3 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ are corrupted when ψ_3 is delivered. $\mathcal{S}$ causes $R^{(s)}$ to immediately abort when it receives an altered ψ_3. $R^{(s)}$ will always abort because the proof in the altered ψ_3 message cannot be correct due to RSig being unforgeable with respect to insider corruption.

6.2.7.5 Indistinguishability under corruptions

This situation occurs when either party is corrupted at a time before the times covered by the previous cases.

The only difference between the normal operation of $\mathcal{S}$ and this case is the secret key used to compute the proofs in messages generated by $\mathcal{S}$, and the mechanism for generating the shared secret keys. Whereas $\mathcal{S}$ normally uses the ephemeral signing keys i and r stored in $\mathcal{N}$ to produce the proofs in the messages, the keys generated by $\mathcal{N}$ might not be used when a party is corrupted before sending its first message. In these cases, $\mathcal{S}$ instead makes use of the long-term secret key of the corrupted party to produce the proofs. Again, these message flows are indistinguishable from real flows due to the anonymity against full key exposure property of RSig. To output the correct session keys, $\mathcal{S}$ extracts the internal state of $\mathcal{N}$ to complete key exchanges with corrupted parties. In all cases, the outputs of the protocol are indistinguishable because the uncorrupted party effectively completes the key exchange honestly.

If both simulated parties are corrupted, then indistinguishability is trivial. $\mathcal{S}$ never generates any messages, and so they cannot be used by $\mathcal{Z}$ to detect simulation. The outputs of corrupted parties are copied to the outputs of the corresponding ideal parties, so this is also not useful to $\mathcal{Z}$.

In all cases of corruption, $\mathcal{S}$ provides the expected memory state for the corrupted party—the set of random coins used to generate ephemeral signing keys, and possibly some shared secret keys (depending on which party is corrupted and when). In all cases, these values are indistinguishable from real values because the parties are effectively simulated honestly.

6.2.7.6 Data from other sessions

Since this proof considers the security of DAKEZ in the EUC model, it must also consider the usefulness of information collected by $\mathcal{Z}$ from other protocol sessions. No information from other sessions can be used to assist $\mathcal{A}$ with the generation of false message flows: ψ_1 is generated using no long-term information, and both ψ_2 and ψ_3 require computation of an SoK bound to the contents of ψ_1. Since RSig is unforgeable with respect to insider corruption, collecting SoKs from other sessions does not allow $\mathcal{A}$ to produce proofs correctly bound to the session under attack. ■

6.3 Proof of Spawn+ Security (Σ)

This section sketches a security proof for Spawn+. The ideal functionality is defined in Section 6.3.1, the real protocol is defined in Section 6.3.2, the security theorem is presented in Section 6.3.3, and the proof sketch begins in Section 6.3.4.

Algorithm 6.6 IDEAL FUNCTIONALITY MODELING SPAWN$^+$, ZDH, AND XZDH'S BEHAVIOR. (Refs: 138 and 160)

Ideal Functionality $\mathcal{F}^+_{1psp\text{-}keia}$

$\mathcal{F}^+_{1psp\text{-}keia}$ proceeds as follows, running on security parameter λ, in the $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$-hybrid model, with parties $\overline{P_1}, \ldots, \overline{P_n}$ and an adversary $\mathcal{S}$. The functionality is parameterized by a non-information oracle $\mathcal{N}$, and an incrimination procedure IncProc. When initializing, $\mathcal{F}^+_{1psp\text{-}keia}$ invokes $\mathcal{N}$ with fresh randomness.

on interaction with $\mathcal{N}$:
  Allow $\mathcal{S}$ to communicate with $\mathcal{N}$ by forwarding messages between them. If at any point $\overline{I}$ or $\overline{R}$ is corrupted while $\mathcal{N}$ has produced local output, send the complete state and local output of $\mathcal{N}$ to $\mathcal{S}$.

on $(\texttt{solicit}, sid, \overline{I}, \Phi, aux)$ **from** $\overline{P} \in \{\overline{P_1}, \ldots, \overline{P_n}\}$:
  if (a `solicit` message seen before) **return**
  if $(\overline{P} \neq \overline{I})$ **return**
  Record that $\overline{I}$ is the initiator
  Mark $\overline{I}$ as "active"
  Send $(\texttt{solicit}, sid, \overline{I}, \Phi)$ to $\mathcal{S}$

on $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi)$ **from** $\overline{P} \in \{\overline{P_1}, \ldots, \overline{P_n}\}$:
  if (an `establish` message seen before) **return**
  if $(\overline{P} \neq \overline{R})$ **return**
  if (initiator not recorded) {
    Resume processing once initiator is recorded
  }
  if ($\overline{I}$ does not match recorded initiator) **return**
  Record that $\overline{R}$ is the responder
  Mark $\overline{R}$ as "active"
  Send $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi)$ to $\mathcal{S}$

on $(\texttt{set-key}, sid)$ **from** $\mathcal{S}$:
  if (a key tuple (sid, κ) has been recorded) **return**
  if (($\overline{I}$ not recorded) || ($\overline{R}$ not recorded)) **return**
  if (($\overline{I}$ is corrupt) **&&** ($\overline{R}$ is corrupt)) **return**
  if (($\overline{I}$ is uncorrupted) **&&** ($\overline{R}$ is uncorrupted)
  ↪ **&&** ($\overline{I}$ is "active")) { $\kappa \xleftarrow{\$} \{0,1\}^\lambda$ }
  else if (IncProc was previously executed) {
  ↪ Let κ denote the local output of IncProc }
  else if ($\overline{I}$ is corrupt) {
  ↪ Let κ denote the local output of $\mathcal{N}$ }
  else { Halt }
  Send $(\texttt{set-key}, sid, \overline{I}, \overline{R}, \kappa)$ to $\overline{R}$
  Record key tuple (sid, κ)

on $(\texttt{finish}, sid, \overline{P})$ **from** $\mathcal{S}$:
  if (no key tuple (sid, κ) has been recorded) **return**
  if ($\overline{I}$ is not "active") **return**
  if ($\overline{P}$ is uncorrupted) {
    Send $(\texttt{set-key}, sid, \overline{I}, \overline{R}, \kappa)$ to $\overline{I}$ and halt
  } **else** {
    Send the state of $\mathcal{N}$ to $\mathcal{S}$
    Wait for $(\texttt{mismatch-key}, sid, k)$ from $\mathcal{S}$
    if $(k = \kappa)$ Halt
    Send $(\texttt{set-key}, sid, \overline{I}, \overline{P}, k)$ to $\overline{I}$ and halt
  }

on $(\texttt{abort}, sid)$ **from** $\mathcal{S}$:
  if (($\overline{I}$ not recorded) $\vee$ ($\overline{R}$ not recorded)) **return**
  Mark $\overline{I}$ as "aborted"
  Send delayed $(\texttt{abort}, sid, \overline{I})$ to $\overline{I}$
  Send delayed $(\texttt{abort}, sid, \overline{R})$ to $\overline{R}$

on $(\texttt{incriminate}, sid)$ **from** $\mathcal{S}$:
  if (IncProc was previously executed) **return**
  if (($\overline{I}$ is "aborted") **&&** ($\overline{R}$ is "active")
  ↪ **&&** ($\overline{R}$ is uncorrupted)) {
    Send $(\texttt{retrievesecret}, \overline{R})$ to $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$
    Retrieve SK_R from $\bar{\mathcal{G}}^{\varphi,n,\mathbb{G},q,g}_{krkro}$
    Execute IncProc$(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$
  }

6.3.1 Ideal Functionality for Spawn+

Unger and Goldberg [UG15] sketched a security proof for Spawn in a GUC-based standard model [UG15]. However, the ideal functionality used in the sketch, $\mathcal{F}^{\text{IncProc}}_{1psp-keia}$, suffers from the same problems discussed in Section 6.2.1, and thus it cannot be used to prove the security of Spawn+. Specifically, $\mathcal{F}^{\text{IncProc}}_{1psp-keia}$ does not capture the notion of contributiveness. Using the same techniques as Section 6.2.1, Algorithm 6.6 constructs a new ideal functionality, $\mathcal{F}^{+}_{1psp-keia}$, that describes a two-flow, single post-specified peer, contributory, and strongly deniable key exchange protocol.

$\mathcal{F}^{+}_{1psp-keia}$ shares similarities with both $\mathcal{F}^{\text{IncProc}}_{1psp-keia}$ and $\mathcal{F}^{+}_{post-keia}$. An initiating party $\bar{I}$ sends a `solicit` message that is answered by a responder $\bar{R}$ with an `establish` message. When both parties are defined, the simulator $\mathcal{S}$ tells $\mathcal{F}^{+}_{1psp-keia}$ to generate a session key and notify $\bar{R}$ using a `set-key` message. $\mathcal{S}$ can then send a `finish` message to send the key to $\bar{I}$. If $\mathcal{S}$ corrupts a party $\bar{P}$, it can cause $\bar{I}$ to erroneously report $\bar{P}$ (rather than $\bar{R}$) as a conversation partner with an arbitrary session key. However, it is not possible for $\mathcal{S}$ to accomplish this without also causing the parties to derive different session keys (leading to termination if high-level protocols incorporate implicit or explicit key confirmation [FGSW16]).[3]

$\mathcal{F}^{+}_{1psp-keia}$ also includes an incriminating abort procedure. Unlike $\mathcal{F}^{+}_{post-keia}$, where $\mathcal{R}$ is forced to abort, $\mathcal{F}^{+}_{1psp-keia}$ forces $\mathcal{I}$ to abort when IncProc is called. The incrimination procedure in $\mathcal{F}^{+}_{1psp-keia}$ also has another important difference: rather than sending the fixed session key to $\mathcal{S}$, it is instead designed to send it directly to $\mathcal{F}^{+}_{1psp-keia}$ as local output on $\mathcal{F}^{+}_{1psp-keia}$'s subroutine tape. This is important for appropriately modeling forward secrecy, as defined in Section 6.2.4. Because $\mathcal{F}^{+}_{1psp-keia}$ can permit an uncorrupted party (specifically, $\bar{R}$) to output a key after IncProc is invoked (unlike $\mathcal{F}^{+}_{post-keia}$, which prevents this possibility), IncProc must not reveal this key to $\mathcal{S}$; doing so would trivially break forward secrecy. Instead, any attack against forward secrecy must exploit the specification of the real protocol by using leaked information from $\mathcal{N}$ or IncProc. Looking ahead, Section 6.5.1 specifies the attack that limits Spawn+ and ZDH to weak forward secrecy, and differentiate the forward secrecy of XZDH.

6.3.2 Spawn+ in the GUC Framework

Spawn+ was originally defined in Section 5.5.1. Algorithm 6.7 redefines Spawn+ in terms of the $\mathcal{F}^{+}_{1psp-keia}$ interface. The protocol is implicitly parameterized with the group $\mathbb{G}$, q, and g. The

[3] This attack also applies to Spawn. The original security proof sketch [Ung15, Section 3.8.4.2] erroneously excludes this possibility.

Algorithm 6.7 SPAWN+ IMPLEMENTED IN THE GUC FRAMEWORK. (Refs: 138 and 140)

Real Protocol | Spawn+

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
  Record that we are the initiator, $\mathcal{I}$
  Retrieve PK_I and SK_I from shared functionality
  Record PK_I, SK_I, sid, and Φ
  Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$
  Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
  Record that we are the responder, $\mathcal{R}$
  Retrieve PK_R and SK_R from shared functionality
  Retrieve PK_I from shared functionality
  Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I)$ **to** $\mathcal{R}$:
  if $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
  Let $t_1 = \mathcal{I} \| \mathcal{R} \| g^i \| PQ_I$
  Compute $\gamma = \text{DREnc}(PK_I, PK_R, g^r \| Q_R, t_1)$
  Let $t_2 = t_1 \| \gamma \| \Phi$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t_2)$
  Compute $k = \text{KDF}((g^i)^r \| Q_k)$
  Erase r and Q_k
  Send $\psi_2 = \mathcal{R} \| \gamma \| \sigma$ to $\mathcal{I}$
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| \gamma \| \sigma)$ **to** $\mathcal{I}$:
  Retrieve PK_P from shared functionality
  Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| PQ_I$
  if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t \| \gamma \| \Phi)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Compute $g^p \| Q_P \leftarrow \text{DRDec}(PK_I, PK_P, SK_I, \Phi, \gamma)$
  Compute $Q_k = \text{QRDecaps}(SQ_I, Q_P)$
  Compute $k = \text{KDF}((g^p)^i \| Q_k)$
  Erase i, SQ_I, and Q_k
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
  Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
  Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

adaptation is very similar to the approach used for DAKEZ in Algorithm 6.3: the parties retrieve the necessary long-term keys from the shared functionality, the initiator broadcasts its initial message using the routing information in *aux*, and the responder replies in the usual manner.

The shared functionality for the proof, $\bar{\mathcal{G}}_{krkro}^{\text{Spawn}^+}$, is defined to be $\bar{\mathcal{G}}_{krkro}^{\text{Spawn}^+,3,\mathbb{G},q,g}$. The key derivation function $\text{KDF}(x)$ is modeled using the first random oracle in $\bar{\mathcal{G}}_{krkro}^{\text{Spawn}^+}$. The second and third random oracles are used to model the hashes in the RSig and DREAD schemes.

6.3.3 Proof Strategy

Theorem 4 states the security theorem for Spawn+.

Theorem 4 | Spawn+ is secure

If the DREAD scheme is sound and IND-CCA2 secure, the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then Spawn+ GUC-realizes $\mathcal{F}_{1psp\text{-}keia}^{+}$ within the erasure $\bar{\mathcal{G}}_{krkro}^{\text{Spawn}^+}$-hybrid model with adaptive security for $\text{IncProc}_{\text{Spawn}^+}$ and non-information oracle $\mathcal{N}_{QRDH}$. (Refs: 140 [a b c d], 148, 150, and 153)

Note that Theorem 4 allows for fully adaptive adversaries due to the use of a non-information oracle, whereas the proof for Spawn only defended against semi-adaptive adversaries [UG15].

The core idea of the proof for Theorem 4 is nearly identical to the security proof for DAKEZ. Algorithm 6.8 shows the incriminating procedure that is used within $\mathcal{F}_{1psp\text{-}keia}^{+}$, $\text{IncProc}_{\text{Spawn}^+}$. $\text{IncProc}_{\text{Spawn}^+}$ simply honestly calculates the second message flow from $\overline{R}$, using SK_R to produce the authenticating proof. The non-information oracle $\mathcal{N}_{QRDH}$ can be directly reused in the proof, since Spawn+ uses the same ephemeral key structure as DAKEZ. Where the details of the sketch are identical to the proof for DAKEZ, the reader is referred to the DAKEZ proof sketch in Section 6.2.

6.3.4 Proof of Theorem 4

(Sketch) The general simulator setup is the same as in Section 6.2. Specifically, a simulator $\mathcal{S}$ is constructed that executes $\mathcal{A}$ internally and simulates parties for $\mathcal{A}$ while interacting with $\mathcal{F}_{1psp\text{-}keia}^{+}$. The first and second flows of Spawn+, as shown in Algorithm 6.7, are referred to as ψ_1 and ψ_2, respectively.

Section 6.3.5 describes the simulator construction, and Section 6.3.6 presents the indistinguishability proof.

Algorithm 6.8 INCRIMINATION PROCEDURE FOR SPAWN+. (Ref: 140)

Subroutine | $\text{IncProc}_{\text{Spawn}^+}(sid, \bar{I}, \bar{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \bar{I}, \bar{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:
Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
Let $t_1 = \text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|PQ_I$
Compute $\gamma = \text{DREnc}(PK_I, PK_R, g^r\|Q_R, t_1)$
Let $t_2 = t_1\|\gamma\|\Phi$
Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t_2)$
Compute $k = \text{KDF}((g^i)^r\|Q_k)$
Compute $\psi = \text{“}R\text{”}\|\gamma\|\sigma$
Send $(\texttt{inc}, sid, \bar{I}, \bar{R}, \psi)$ to $\mathcal{S}$
Locally output k
Halt

6.3.5 Simulator Construction

6.3.5.1 General handling of $\mathcal{A}$

As in Section 6.2.6, $\mathcal{S}$ allows $\mathcal{A}$ and $\mathcal{Z}$ to communicate, delays messages delayed by $\mathcal{A}$, corrupts ideal parties when $\mathcal{A}$ corrupts the corresponding simulated parties, and replicates output from corrupted parties.

6.3.5.2 Initialization

When $\mathcal{S}$ first initializes, it sends a $(\texttt{setup}, \mathbb{G}, q, g)$ message to $\mathcal{N}$ through $\mathcal{F}^+_{\textit{post-keia}}$, and waits to receive a $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ message in response. The group details sent by $\mathcal{S}$ correspond to the protocol instantiation attacked by $\mathcal{A}$. $\mathcal{S}$ makes a note of the ephemeral keys in the `exchange` message for later use in the simulation.

6.3.5.3 Receipt of `solicit` message from $\mathcal{F}^+_{\textit{1psp-keia}}$

When $\mathcal{S}$ receives $(\texttt{solicit}, sid, \bar{I}, \Phi_I)$ from $\mathcal{F}^+_{\textit{1psp-keia}}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ computes $\psi_1 = \text{“}I\text{”} \parallel g^i \parallel PQ_I$ using the g^i

and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ records the value Φ_I for later use.

6.3.5.4 Receipt of `establish` message from $\mathcal{F}^+_{1psp\text{-}keia}$

When $\mathcal{S}$ receives an establishment message $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi_R)$ from $\mathcal{F}^+_{1psp\text{-}keia}$, it checks to see the circumstances of the simulated ψ_1 message transmission. $\mathcal{S}$ parses ψ_1 to recover g^i and PQ_I. If ψ_1 is not of the correct format, or if the asserted identity is not $I^{(s)}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting `abort` to $\overline{R}$ immediately, and withholds the `abort` message to $\overline{I}$. Otherwise, $\mathcal{S}$ constructs a message ψ_2 from $R^{(s)}$ in response to ψ_1 and causes $\mathcal{F}^+_{1psp\text{-}keia}$ to record a shared secret key. The mechanism for constructing ψ_2 depends on how ψ_1 was generated:

- If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message ψ_1', but $\psi_1 \neq \psi_1'$, then $\mathcal{A}$ has altered the message in transit. $\mathcal{S}$ constructs ψ_2 through the use of IncProc. $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, but withholds delivery of the resulting `abort` messages to $\overline{I}$ and $\overline{R}$. It then sends $(\texttt{incriminate}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, causing an instance of IncProc to be invoked. Using the values parsed from ψ_1, $\mathcal{S}$ sends $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi_R, g^i, PQ_I)$ to IncProc, causing IncProc to send the message $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi_2)$ in response. IncProc will then privately send the session key to $\mathcal{F}^+_{1psp\text{-}keia}$.
- Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a ciphertext and forged proof. The method for choosing the ephemeral keys contained within the ciphertext and the keys used to forge the proof depends on the environment. Given a choice of ephemeral keys for $R^{(s)}$, g^r and Q_R, $\mathcal{S}$ produces a ciphertext $\gamma = \text{DREnc}(PK_I, PK_R, g^r \| Q_R, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I)$. $\mathcal{S}$ then produces a proof σ_R with tag $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I \| \gamma \| \Phi_R$. $\mathcal{S}$ selects ephemeral keys and produces the proof in the following way:
  - If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ uses the values of g^r and Q_R retrieved from the non-information oracle $\mathcal{N}$ to produce γ. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending $(\texttt{prove}, 1, \{PK_I, PK_R, g^i\}, t)$ to $\mathcal{N}$ and waiting for a message $(\texttt{proof}, \sigma_R)$ in response.
  - If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I)$ to $\mathcal{N}$. Since $\overline{I}$ is corrupted, $\mathcal{F}^+_{1psp\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ uses the newly generated values g^r and Q_R as the ephemeral keys to be contained within γ. $\mathcal{S}$ uses its access to corrupt $\overline{I}$

to retrieve SK_I from $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ with a `retrievesecret` message. $\mathcal{S}$ then forges the proof using $\sigma_R = \text{RSig}(PK_I, SK_I, \{PK_I, PK_R, g^i\}, t)$.

Given γ and σ_R, $\mathcal{S}$ constructs $\psi_2 =$ "R" $\| \gamma \| \sigma_R$.

$\mathcal{S}$ then sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$. $\mathcal{S}$ sends (`set-key`, sid) to $\mathcal{F}^+_{1psp\text{-}keia}$, causing $\mathcal{F}^+_{1psp\text{-}keia}$ to record a shared secret key and causing $\overline{R}$ to output that key. If IncProc was used, $\mathcal{F}^+_{1psp\text{-}keia}$ will use its output as the key. Otherwise, $\mathcal{F}^+_{1psp\text{-}keia}$ will use the output from $\mathcal{N}$ if $I^{(s)}$ is corrupted, or a fresh random value if $I^{(s)}$ is uncorrupted.

6.3.5.5 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract "P", the ciphertext γ, and the proof σ_P. If ψ_2 is not of the correct form, or if σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, then $\mathcal{S}$ sends (`abort`, sid) to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting `abort` message to $\overline{I}$ immediately, and withholds the `abort` message to $\overline{R}$.

If $I^{(s)}$ has previously broadcast a message ψ_1 and ψ_2 is valid, then $\mathcal{S}$ must cause $\overline{I}$ to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. The key depends on the state of the simulation:

- If $\mathcal{A}$ has previously corrupted $P^{(s)}$, then $\mathcal{S}$ must have previously corrupted $\overline{P}$ (since $\mathcal{S}$ corrupts ideal parties corresponding to simulated parties corrupted by $\mathcal{A}$). $\mathcal{S}$ issues a (`finish`, sid, $\overline{P}$) message to $\mathcal{F}^+_{1psp\text{-}keia}$. Since $\overline{P}$ is corrupted, $\mathcal{F}^+_{1psp\text{-}keia}$ sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ recovers the ephemeral keys i and SQ_I used to generate ψ_1 from $\mathcal{N}$. $\mathcal{S}$ retrieves SK_P from $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ using a `retrievesecret` message, then uses this key to decrypt γ, recovering g^p and Q_P. $\mathcal{S}$ computes $k = \text{KDF}((g^p)^i \| \text{QRDecaps}(SQ_I, Q_P))$ and sends (`mismatch-key`, sid, k) to $\mathcal{F}^+_{1psp\text{-}keia}$, causing $\overline{I}$ to output key k and partner identity $\overline{P}$ with overwhelming probability.
- Otherwise, $P^{(s)}$ is uncorrupted. Because RSig is unforgeable with respect to insider corruption, the only way for σ_P to be valid in this situation is if the sender of the message knows the long-term private key of one of the parties, or the ephemeral key of $I^{(s)}$. Additionally, σ_P must have been computed during this session because its validity depends on ψ_1. Due to the hardness of the discrete log problem in the group (guaranteeing the secrecy of the keys) and the uniqueness of ψ_1, this is only possible if $P^{(s)} = R^{(s)}$ and ψ_2 was previously generated by $\mathcal{S}$. $\mathcal{S}$ sends (`finish`, sid, $\overline{R}$) to $\mathcal{F}^+_{1psp\text{-}keia}$, causing $\overline{I}$ to emit an appropriate session key that is shared with $\overline{R}$.

6.3.5.6 Transmission of ψ_1 by corrupted $I^{(s)}$

When $\mathcal{S}$ has not yet received a `solicit` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $I^{(s)}$ to issue message ψ_1, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\bar{I}$ to send $(\texttt{solicit}, sid, \bar{I}, \perp, \perp)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, but ignores the resulting `solicit` message sent by $\mathcal{F}^{+}_{1psp\text{-}keia}$.

6.3.5.7 Transmission of ψ_2 by corrupted $R^{(s)}$

When $\mathcal{S}$ has not yet received an `establish` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $R^{(s)}$ to issue message ψ_2, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{R}$ to send $(\texttt{establish}, sid, \bar{I}, \overline{R}, \perp)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, but ignores the resulting `establish` message sent by $\mathcal{F}^{+}_{1psp\text{-}keia}$.

6.3.5.8 Constructing state for corrupted parties

When $\mathcal{A}$ corrupts a party in the simulated environment, $\mathcal{S}$ corrupts the corresponding party in the ideal environment. If $\mathcal{A}$ causes corrupted parties to output values, $\mathcal{S}$ outputs these values from the corresponding ideal parties. In addition, $\mathcal{S}$ must provide $\mathcal{A}$ with a simulated historical state for corrupted parties.

If $\mathcal{A}$ corrupts the party known as $I^{(s)}$ after a `solicit` message has been received, but before $I^{(s)}$ has received ψ_2, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to provide the random coins i, and SQ_I used to construct ψ_1. If $I^{(s)}$ already received ψ_2, then $\mathcal{S}$ uses its corruption of $\bar{I}$ to provide the session key k that $\bar{I}$ already output.

If $\mathcal{A}$ corrupts the party known as $R^{(s)}$ after it has already received ψ_1, then $\mathcal{S}$ uses its corruption of $\overline{R}$ to provide the session key k that $\overline{R}$ already output.

6.3.6 Proof of Indistinguishability

The proof of indistinguishability is similar to the proof described in Section 6.2.7. It must be shown that flows and memory states are indistinguishable in all cases.

6.3.6.1 The honest case

This situation occurs when $\mathcal{A}$ does not corrupt $I^{(s)}$ or $R^{(s)}$ until after the session concludes, or alter any message flows. The proof of this case is the same as the honest case in Section 6.2.7;

since the protocol is executed honestly (with the exception of proof generation), $\mathcal{Z}$ would need to break the anonymity against full key exposure of RSig to identify that the proofs were forged, or $\mathcal{Z}$ would need to break the CDH assumption within $\mathbb{G}$ to identify that the shared session key was randomly generated.

6.3.6.2 Alteration of ψ_1

This situation occurs when ψ_1 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ is corrupted when ψ_1 is delivered. The proof of this case is the same as in Section 6.2.7, except that $\mathcal{Z}$ is given even less information when subsequently corrupting $R^{(s)}$; both the ψ_2 message and session key generated by IncProc are effectively generated honestly, so the messages and output from $R^{(s)}$ are indistinguishable from a real protocol execution. $I^{(s)}$ aborts as expected if ψ_2 is delivered, since RSig is unforgeable with respect to insider corruption.

6.3.6.3 Alteration of ψ_2

This situation occurs when ψ_2 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $P^{(s)}$ (the party named as the communication partner in ψ_2) are corrupted when ψ_2 is delivered. $\mathcal{S}$ causes $I^{(s)}$ to immediately abort when it receives an altered ψ_2. As mentioned previously, $I^{(s)}$ will always abort because the proof in the altered ψ_2 message cannot be correct due to the unforgeability with respect to insider corruption of RSig.

6.3.6.4 Indistinguishability under corruptions

This situation occurs when either party is corrupted at a time before the times covered by the previous cases. The proof of this case is nearly the same as in Section 6.2.7. When both $I^{(s)}$ and $R^{(s)}$ are corrupted, indistinguishability is trivial because $\mathcal{S}$ can directly replicate the behavior of $\mathcal{A}$. With only a single corrupted party, the other effectively performs the protocol honestly. The exception is when $I^{(s)}$ is corrupted but $R^{(s)}$ is not, in which case $\mathcal{S}$ forges the proof in ψ_2 using the long-term key of $\bar{I}$. This forgery is undetectable by $\mathcal{Z}$ due to the anonymity against full key exposure of RSig, even if $\mathcal{A}$ corrupts all parties after the session. All memory states are consistent with honest execution, as mentioned above.

The only major difference with Section 6.2.7 is that $\mathcal{A}$ can cause $I^{(s)}$ to accept the wrong communication partner without aborting the protocol by leaving $I^{(s)}$ and $R^{(s)}$ uncorrupted,

corrupting $P^{(s)}$, and replacing $R^{(s)}$'s ψ_2 message with an honestly generated message from $P^{(s)}$.[4] In this case, $\mathcal{F}^{+}_{1psp\text{-}keia}$ grants $\mathcal{S}$ access to the internal state of $\mathcal{N}$. $\mathcal{S}$ uses this state to compute $\bar{I}$'s output honestly, with the exception of using $\bar{P}$'s long-term secret key to decrypt the DREAD ciphertext. Due to the soundness of the DREAD scheme, this process results in the same key that would have been computed by an honestly behaving $I^{(s)}$, so the key must be the one that $\mathcal{Z}$ expects. Revealing the state of $\mathcal{N}$ and allowing $\mathcal{S}$ to completely dictate the output of $\bar{I}$ is not a problem in this case, since $\mathcal{A}$ effectively controls one of the conversation partners from the perspective of $I^{(s)}$. Leaking $\mathcal{N}$'s state to $\mathcal{S}$ models that the adversary has access to the key shared with $\bar{I}$, and it has full control over the value of the key (since it controls the second ephemeral key contribution).

6.3.6.5 Data from other sessions

As in Section 6.2.7, no information from other sessions can be used to assist $\mathcal{A}$ with the generation of false message flows: ψ_1 is generated using no long-term information, and ψ_2 requires computation of an SoK bound to the contents of ψ_1. Since RSig is unforgeable with respect to insider corruption, collecting SoKs from other sessions does not allow $\mathcal{A}$ to produce proofs correctly bound to the session under attack. Additionally, the IND-CCA2 security of the DREAD scheme implies that it is also non-malleable [BDPR98]. This non-malleability prevents $\mathcal{A}$ from modifying DREAD ciphertexts from other sessions to bind them to the session under attack without corrupting one of the parties capable of decrypting the ciphertext. ■

6.4 Proof of ZDH Security Σ

It is possible to prove the security of ZDH in nearly the same environment as Spawn+, since the protocols share most properties. In fact, ZDH can GUC-realize the same ideal functionality $\mathcal{F}^{+}_{1psp\text{-}keia}$ that is GUC-realized by Spawn+. Algorithm 6.9 defines ZDH (previously defined in Section 5.5) in terms of the $\mathcal{F}^{+}_{1psp\text{-}keia}$ interface. The security theorem for ZDH is as follows:

[4] ^ As mentioned in Section 6.3.1, key confirmation can be used in higher-level protocols to alleviate this weakness.

Algorithm 6.9 ZDH IMPLEMENTED IN THE GUC FRAMEWORK. (Refs: 146, 154 [a] [b], and 156)

Real Protocol | ZDH

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
  Record that we are the initiator, $\mathcal{I}$
  Retrieve PK_I and SK_I from $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$
  Record PK_I, SK_I, sid, and Φ
  Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$
  Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux to route

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
  Record that we are the responder, $\mathcal{R}$
  Retrieve PK_R and SK_R from $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$
  Retrieve PK_I from $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$
  Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I)$ **to** $\mathcal{R}$:
  if $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
  Compute $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$
  Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
  Let $t = \mathcal{I} \| \mathcal{R} \| g^i \| g^r \| PQ_I \| Q_R \| \Phi$
  Compute $mac = \text{MAC}(M_k, t)$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
  Erase r, Q_k, κ, and M_k
  Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| mac \| \sigma$ to $\mathcal{I}$
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| g^p \| Q_P \| mac \| \sigma)$ **to** $\mathcal{I}$:
  Retrieve PK_P from $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$
  Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| \Phi$
  if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Compute $Q_k = \text{QRDecaps}(SQ_I, Q_P)$
  Compute $\kappa = \text{KDF}_1((g^p)^i \| (g^p)^{SK_I} \| Q_k)$
  Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
  Compute $mac' = \text{MAC}(M_k, t)$
  if $(mac \neq mac')$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Erase i, SQ_I, Q_k, κ, and M_k
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
  Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
  Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 6.10 INCRIMINATION PROCEDURE FOR ZDH. (Ref: 150)

Subroutine | $\text{IncProc}_{\text{ZDH}}(sid, \bar{I}, \bar{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \bar{I}, \bar{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:
Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
Compute $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$
Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
Let $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^r \| PQ_I \| Q_R \| \Phi$
Compute $mac = \text{MAC}(M_k, t)$
Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
Compute $\psi = \text{“}R\text{”} \| g^r \| Q_R \| mac \| \sigma$
Send $(\texttt{inc}, sid, \bar{I}, \bar{R}, \psi)$ to $\mathcal{S}$
Locally output k
Halt

Theorem 5 | ZDH is secure

If the MAC is weakly unforgeable under chosen message attack [BKR00], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then ZDH GUC-realizes $\mathcal{F}^+_{1psp\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$-hybrid model with partially adaptive security for $\text{IncProc}_{\text{ZDH}}$ and non-information oracle $\mathcal{N}_{ZDH}$. (Refs: 148[ab], 150[abcd], 152, 153[ab], 157, and 158)

There are a few key differences between Theorem 5 and Theorem 4. ZDH uses a protocol-specific incrimination procedure, $\text{IncProc}_{\text{ZDH}}$, that simply provides an honest generation of the second message flow using $\bar{R}$'s long-term secret key. $\mathcal{N}_{ZDH}$ is similar to $\mathcal{N}_{QRDH}$, but also enables calls to MAC with the shared key M_k, and supports adding extra DH secrets into the KDF. The phrase “partially adaptive” in Theorem 5 means that $\mathcal{S}$ may corrupt $\bar{I}$ at any time, but $\mathcal{S}$ may only corrupt $\bar{R}$ after a `finish` message has been received by $\mathcal{F}^+_{1psp\text{-}keia}$. In the ideal setting, this means that $\mathcal{S}$ may not send an `establish` message from a corrupted party. In the real setting, this means that $\mathcal{A}$ cannot transmit a response message ψ_2 from a corrupted party. This restriction intentionally prevents the protocol from achieving online deniability for $\mathcal{R}$ in order to allow ZDH to GUC-realize $\mathcal{F}^+_{1psp\text{-}keia}$. The shared functionality for the proof, $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$, is defined to be $\bar{\mathcal{G}}^{\text{ZDH},4,\mathbb{G},q,g}_{krkro}$. The first three random oracles in $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ are used to model KDF_1, KDF_2, and KDF_3. The fourth random oracle models the hash in the RSig scheme.

Algorithm 6.11 NON-INFORMATION ORACLE FOR ZDH. (Ref: 150)

Subroutine | $\mathcal{N}_{ZDH}$

$\mathcal{N}_{ZDH}$ copies the `setup` and `prove` commands from $\mathcal{N}_{QRDH}$ (Algorithm 6.5), slightly modifies its `complete` command, and introduces two new commands.

function KDF-VAL(C, r, Q_k)
  $\gamma \leftarrow \emptyset$
  for each ($c \in C$) {
    if (($c \notin \mathbb{G}$) || (c is identity element)) Halt
    $\gamma \leftarrow \gamma \| c^r$
  }
  return $\text{KDF}_1(\gamma \parallel Q_k)$

on (`setup`, $\mathbb{G}, q, g$) **from** $\mathcal{M}$:
  Handle the message as in $\mathcal{N}_{QRDH}$

on (`prove`, p, S, m) **from** $\mathcal{M}$:
  Handle the message as in $\mathcal{N}_{QRDH}$

on (`complete`, ok, α, β, C) **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (already output a key) **return**
  if (ok is TRUE) {
    Compute k = KDF-VAL($\{g^i\} \cup C$, r, Q_k)
  } **else** {
    Generate new $r \xleftarrow{\$} \mathbb{Z}_q$
    Generate new $(Q_R, Q_k) \leftarrow$ QREncaps(β)
    Compute k = KDF-VAL($\{\alpha\} \cup C$, r, Q_k)
  }
  Locally output k

on (`authenticate`, C, m) **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  Compute k = KDF-VAL($\{g^i\} \cup C$, r, Q_k)
  Compute $mac = \text{MAC}(\text{KDF}_2(k), m)$
  Send (`authentication`, mac) to $\mathcal{M}$

on (`verify`, C, m, mac) **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  Compute k = KDF-VAL($\{g^i\} \cup C$, r, Q_k)
  Compute $mac' = \text{MAC}(\text{KDF}_2(k), m)$
  if ($mac' = mac$) { Send (`verified`, TRUE) to $\mathcal{M}$ }
  else { Send (`verified`, FALSE) to $\mathcal{M}$ }

The proof sketch of Theorem 5 is nearly identical to the one for Spawn+, since the protocols behave similarly in all applicable scenarios. While $\mathcal{F}^{+}_{1psp\text{-}keia}$ provides some additional features that are not necessary for the proof of Theorem 5, the proof sketch avoids defining a new ideal functionality in order to clearly show the security relationship between Spawn+ and ZDH.

Algorithm 6.10 shows the incrimination procedure for ZDH, which is simply an honest generation of a response ψ_2 using $\mathcal{R}$'s long-term secret key to produce the proof. Algorithm 6.11 shows the non-information oracle used for the proof. $\mathcal{N}_{ZDH}$ is similar to $\mathcal{N}_{QRDH}$, except that it also adds a facility for the simulator to generate and verify a MAC using the shared key derived from the exchange. $\mathcal{N}_{ZDH}$ also permits the caller $\mathcal{M}$ to pass in a set of additional DH public contributions, C, that are combined with the responder's ephemeral key, r, as part of the input to KDF_1. For ZDH, C contains only the initiator's long-term public key, PK_I.

It is clear that $\mathcal{N}_{ZDH}$ is still a non-information oracle, despite the addition of the new MAC methods and the introduction of the C parameter. As in $\mathcal{N}_{QRDH}$, $\mathcal{M}$ is never given enough information to compute g^{ir} (when ok is TRUE) or α^r (when ok is FALSE). $\mathcal{M}$ is also not given enough information to compute Q_k. Because KDF_1 is modeled by a random oracle, the local output from `complete` is indistinguishable from random without access to these values, which are used as input to KDF_1 in KDF-VAL. This holds true even if $\mathcal{M}$ is able to compute the other values concatenated into γ using knowledge of the secrets associated with keys in C, since it cannot compute the entire input to KDF_1. For the same reason, the outputs of KDF_1 and KDF_2 that are passed to MAC in the `authenticate` and `verify` handlers is independent of i, r, and Q_k. Note that the non-information property does *not* depend on the security of MAC, since the MAC key is already computationally independent of the secrets and $\mathcal{M}$ knows m.

The proof proceeds as follows:

(Sketch) The proof of Theorem 5 is nearly identical to the proof of Theorem 4 in Section 6.3. For this reason, the sketch only highlights the differences in simulator construction. Section 6.4.1 constructs the simulator and Section 6.4.2 demonstrates indistinguishability.

6.4.1 Simulator Construction

The simulator $\mathcal{S}$ behaves exactly as in Section 6.3.5 except that in the proof for Theorem 5, all references to $\tilde{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ are replaced by references to $\tilde{\mathcal{G}}^{\text{ZDH}}_{krkro}$.

6.4.1.1 Receipt of `establish` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$

This case is identical to the case in Section 6.3.5, except for the mechanism for constructing ψ_2:

- If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message ψ_1', but $\psi_1 \neq \psi_1'$, then $\mathcal{S}$ behaves as in Section 6.3.5. The only difference is that IncProc returns a properly constructed ψ_2 message for ZDH.
- Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a forged proof σ_R with tag t = "I"‖"R"‖g^i‖g^r‖PQ_I‖Q_R‖Φ_R. The method for choosing the ephemeral keys (g^r and Q_R) and the keys used to forge the proof depends on the environment:
  - If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ uses the values of g^r and Q_R retrieved from the non-information oracle $\mathcal{N}$. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending (`prove`, 1, $\{g^I, g^R, g^i\}, t$) to $\mathcal{N}$ and waiting for a message (`proof`, σ_R) in response. $\mathcal{S}$ also requests a MAC by sending (`authenticate`, $\{PK_I\}, t$) to $\mathcal{N}$ and waiting for a message (`authentication`, mac) in response.
  - If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message (`complete`, FALSE, $g^i, PQ_I, \{PK_I\}$) to $\mathcal{N}$. Since $\bar{I}$ is corrupted, $\mathcal{F}^+_{1psp\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ uses the newly generated values g^r and Q_R as the ephemeral keys for ψ_2. $\mathcal{S}$ uses its access to corrupt $\bar{I}$ to retrieve SK_I from $\tilde{\mathcal{G}}^{\text{ZDH}}_{krkro}$ with a `retrievesecret` message. $\mathcal{S}$ forges the proof using $\sigma_R = \text{RSig}(PK_I, SK_I, \{g^I, g^R, g^i\}, t)$. $\mathcal{S}$ also uses the state of $\mathcal{N}$ to compute mac. $\mathcal{S}$ computes $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and then $mac = \text{MAC}(M_k, t)$.

  Given a choice of ephemeral keys, mac, and σ_R, $\mathcal{S}$ constructs ψ_2 = "R" ‖ g^r ‖ Q_R ‖ mac ‖ σ_R.

As in Section 6.3.5, $\mathcal{S}$ sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$, and then sends (`set-key`, sid) to $\mathcal{F}^+_{1psp\text{-}keia}$.

6.4.1.2 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract "P", g^p, Q_P, mac_P, and the proof σ_P. If ψ_2 is not of the correct form, σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, or if $P^{(s)}$ is corrupted, then $\mathcal{S}$ sends (`abort`, sid) to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting `abort` message to $\bar{I}$ immediately, and withholds the `abort` message to $\bar{R}$.

If $I^{(s)}$ has previously broadcast a message ψ_1, ψ_2 is valid, and $P^{(s)}$ is uncorrupted, then $\mathcal{S}$ must cause $\bar{I}$ to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. $\mathcal{S}$ proceeds under the assumption that $P^{(s)} = R^{(s)}$ (the proof will later argue that this must be

the case). $\mathcal{S}$ uses $\mathcal{N}$ to check the validity of mac_P by sending $(\texttt{verify}, \{PK_I\}, t, mac_P)$, where tag $t = \text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^p\|PQ_I\|Q_P\|\Phi_I$. If $\mathcal{S}$ receives $(\texttt{verified}, \text{FALSE})$ in response, then it aborts $\bar{I}$ in the same manner as above. Otherwise, $\mathcal{S}$ sends $(\texttt{finish}, sid, \overline{R})$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, causing $\bar{I}$ to emit a session key shared with $\overline{R}$.

6.4.1.3 Transmission of ψ_2 by corrupted $R^{(s)}$

$\mathcal{A}$ is not permitted to transmit ψ_2 messages from corrupted parties, so this case is no longer needed.

6.4.2 Proof of Indistinguishability

The proof of indistinguishability given in Section 6.3.6 also applies here with several changes. The proof no longer needs to consider corruptions of the responder before transmission of ψ_2, so these cases can be ignored. As in Section 6.3.6, the anonymity against full key exposure of RSig prevents $\mathcal{Z}$ from detecting forgeries of σ_P in ψ_2 by $\mathcal{S}$, even if $\mathcal{A}$ corrupts all parties after the session. Similarly, the CDH assumption on the underlying group prevents $\mathcal{Z}$ from distinguishing the output key in the honest case, and the unforgeability of RSig with respect to insider corruption prevents $\mathcal{Z}$ from producing forged proofs in ψ_2, even with information collected from other sessions.

The only significant difference with the proof in Section 6.3.6 is indistinguishability in the case of an altered ψ_2 received by an uncorrupted $I^{(s)}$. Due to the unforgeability of RSig with respect to insider corruption, any modification to ψ_2 must include a proof σ_P naming a corrupted party $P^{(s)} \neq R^{(s)}$. In this case, $\mathcal{S}$ always aborts $\bar{I}$. This procedure always matches the simulated environment because ψ_2 must be invalid. If ψ_2 includes a valid proof σ_P and $I^{(s)}$ does *not* abort, then this implies that mac_P is also valid. Due to the weak unforgeability under chosen message attack of MAC, this is only possible if $\mathcal{A}$ knows the MAC key derived from a secret shared between $I^{(s)}$ and $P^{(s)}$, which in turn is only possible (due to the security of the key derivation functions and the hardness of the CDH problem in the underlying group) if $\mathcal{A}$ honestly generated a response from $P^{(s)}$. An honest generation of this type is not permitted due to the “partially adaptive” constraint in Theorem 5, because it corresponds to the completion of the protocol by a corrupted responder. Consequently, $\mathcal{A}$ cannot possibly cause $I^{(s)}$ to output a key when altering ψ_2. ■

6.5 Proof of XZDH Security (Σ)

This section sketches a security proof for the final new DAKE, XZDH. Section 6.5.1 discusses how the forward secrecy of XZDH is defined in the GUC model. Section 6.5.2 presents the formalized security theorem. The proof sketch begins in Section 6.5.3

6.5.1 Forward Secrecy of DAKEZ, Spawn+, and ZDH

Chapter 5 previously claimed that DAKEZ has strong forward secrecy, and that the purpose of XZDH is to improve the forward secrecy properties of ZDH. Using the framework established so far in this chapter, it is now possible to precisely distinguish between these forward secrecy levels from the perspective of the GUC-based proof sketches.

Section 6.2.4 defined strong and weak forward secrecy in terms of ideal GUC processes. In these terms, the proof of Theorem 3 shows that DAKEZ offers strong forward secrecy. Consider the simulator $\mathcal{S}$ interacting with $\mathcal{F}^{+}_{post\text{-}keia}$ as sketched in Section 6.2.6. In order to break strong forward secrecy, $\mathcal{Z}$ must use $\mathcal{S}$ to collect enough information from $\mathcal{N}_{QRDH}$ or $\text{IncProc}_{\text{DAKEZ}}$ to distinguish a key output by $\overline{I}$ or $\overline{R}$ from random without corrupting the entity or its partner. The definition of $\mathcal{F}^{+}_{post\text{-}keia}$ guarantees that no keys can be output when $\text{IncProc}_{\text{DAKEZ}}$ is invoked, so the incrimination procedure cannot be used to gather information. Moreover, the only time that $\mathcal{S}$ sends a `deliver` message when a party and its partner are uncorrupted is when both $\overline{I}$ and $\overline{R}$ are uncorrupted, the key was generated by $\mathcal{N}_{QRDH}$, and $\mathcal{N}_{QRDH}$'s proposed exchange was accepted. By the non-information property, $\mathcal{N}_{QRDH}$ yields no information about this key.

In contrast, $\mathcal{F}^{+}_{1psp\text{-}keia}$ only guarantees weak forward secrecy, which prevents $\mathcal{S}$ from aborting the session. IncProc is not useful to $\mathcal{Z}$ because it requires an `abort` command to be invoked. By definition, $\mathcal{N}$ is also not useful, even if its proposed exchange is rejected. Proving Theorem 4 and Theorem 5 therefore shows that Spawn+ and ZDH have (at least) weak forward secrecy.

If Spawn+ and ZDH do not offer strong forward secrecy, then there should exist an attack in which $\mathcal{Z}$ can issue instructions to $\mathcal{S}$ to learn information about a key output by $\overline{P}$ with partner $\overline{P'}$ without corrupting either $\overline{P}$ or $\overline{P'}$ during the corresponding session. Indeed, such an attack exists when $\mathcal{S}$ is permitted to abort $\mathcal{F}^{+}_{1psp\text{-}keia}$. For simplicity, this section considers only the ZDH case—a similar attack works against Spawn+. The attack, which takes place in the setting of Theorem 5 and is the idealized equivalent of the one discussed in Section 5.6, proceeds as follows:

1. $\mathcal{Z}$ instructs $\overline{I}$ to send a `solicit` message to $\mathcal{F}^{+}_{1psp\text{-}keia}$. This causes $\mathcal{S}$ to simulate a message ψ_1 from $I^{(s)}$.

2. $\mathcal{Z}$ generates $z \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_Z, SQ_Z) \leftarrow \text{QRGen}()$. $\mathcal{Z}$ then instructs $\mathcal{S}$ to modify ψ_1 to be "I" $\| g^z \| PQ_Z$.

3. $\mathcal{Z}$ instructs $\overline{R}$ to send an `establish` message to $\mathcal{F}^{+}_{1psp\text{-}keia}$. This causes $\mathcal{S}$ to abort the protocol, interact with $\text{IncProc}_{\text{ZDH}}$, and simulate the resulting ψ_2 = "R" $\| g^r \| Q_R \| mac \| \sigma_R$ message from $R^{(s)}$. $\overline{R}$ receives a `set-key` message identifying partner $\overline{I}$ with a key k generated by $\text{IncProc}_{\text{ZDH}}$. Note that $\mathcal{S}$ does not learn k, since $\text{IncProc}_{\text{ZDH}}$ sends it directly to $\mathcal{F}^{+}_{1psp\text{-}keia}$.

4. $\mathcal{Z}$ instructs $\mathcal{S}$ to deliver ψ_2 to $I^{(s)}$, which causes $\mathcal{S}$ to issue an `abort` command and deliver an `abort` message to $\overline{I}$. The session has now completed.

5. $\mathcal{Z}$ causes $\overline{I}$ to be corrupted, revealing secret key I. $\mathcal{Z}$ can now compute $(g^r)^z$, $(g^r)^I$, and $\text{QRDecaps}(SQ_Z, Q_R)$, which allows it to derive k. $\mathcal{Z}$ can distinguish k from random, even though neither $\overline{I}$ nor $\overline{R}$ were corrupted during the session that $\overline{R}$ outputted k.

6.5.2 XZDH Security Theorem

The only difference between XZDH and ZDH is the introduction of signed prekeys in XZDH. Nonetheless, the presence of signed prekeys necessitates some changes to the security model.

Signed prekeys persist across protocol sessions, so they must be modeled as part of the shared functionality. Algorithm 6.12 depicts $\bar{\mathcal{G}}^{XZDH}$, a shared functionality that combines $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ with a mechanism for sharing signed prekeys. This mechanism is essentially another instance of a key registration with knowledge functionality, $\bar{\mathcal{G}}^{\varphi}_{krk}$, except with the ability for keys to be replaced with new ones, and distribution of digital signatures produced using the long-term keys.

XZDH was originally defined in Section 5.6. Algorithm 6.13 depicts the XZDH protocol following the $\mathcal{F}^{+}_{1psp\text{-}keia}$ interface. The protocol is nearly identical to Algorithm 6.9, except that it incorporates signed prekey distribution. Differences between Algorithm 6.13 and Algorithm 6.9 are shaded in the protocol definition.

The security theorem for XZDH is as follows:

Theorem 6 | XZDH is secure

If the MAC is weakly unforgeable under chosen message attack [BKR00], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then XZDH GUC-realizes $\mathcal{F}^{+}_{1psp\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{XZDH}$-hybrid model with partially adaptive security for $\text{IncProc}_{\text{XZDH}}$ and non-information oracle $\mathcal{N}_{ZDH}$. (Refs: 157[a b], 158[a b c], and 502)

Algorithm 6.12 SHARED FUNCTIONALITY SUPPORTING SIGNED PREKEYS USED BY XZDH. (Ref: 154)

Shared Functionality $\bar{\mathcal{G}}^{XZDH}$

$\bar{\mathcal{G}}^{XZDH}$ copies all behavior and commands supported by $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ (i.e., it handles `register`, `retrieve`, `retrievesecret`, and `ro` messages identically). In addition, it also supports the following commands:

on (`spk-refresh`) **from** $\mathcal{P}$:
  if (no tuple $(\mathcal{P}, PK, SK)$ is recorded) **return**
  if ($\mathcal{P}$ is corrupt) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}, \cdot, \cdot, \cdot)$ is recorded) {
    Delete t
  }
  $r \xleftarrow{\$} \mathbb{Z}_q$
  Record tuple $(\texttt{spk}, \mathcal{P}, g^r, r, \text{Sig}(PK, SK, g^r))$

on (`spk-corrupt-refresh`, r) **from** $\mathcal{P}$:
  if (no tuple $(\mathcal{P}, PK, SK)$ is recorded) **return**
  if ($\mathcal{P}$ is uncorrupted) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}, \cdot, \cdot, \cdot)$ is recorded) {
    Delete t
  }
  Record tuple $(\texttt{spk}, \mathcal{P}, g^r, r, \text{Sig}(PK, SK, g^r))$

on (`spk-retrieve`, $\mathcal{P}'$) **from** $\mathcal{P}$:
  if (tuple $t = (\texttt{spk}, \mathcal{P}', pk, \cdot, \xi)$ is recorded) {
    Send $(\texttt{spk}, \mathcal{P}', pk, \xi)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{spk}, \mathcal{P}', \perp, \perp)$ to $\mathcal{P}$
  }

on (`spk-retrievesecret`, $\mathcal{P}'$) **from** $\mathcal{P}$:
  if (($\mathcal{P}$ is honest) $\wedge$ ($\mathcal{P}$'s code $\notin \varphi$)) **return**
  if (($\mathcal{P}$ is corrupt) $\wedge$ ($\mathcal{P} \neq \mathcal{P}'$)) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}', pk, sk, \xi)$ is recorded) {
    Send t to $\mathcal{P}$
  }

Algorithm 6.13 XZDH IMPLEMENTED IN THE GUC FRAMEWORK. Shaded lines indicate differences from the ZDH implementation in Algorithm 6.9. (Refs: 154 [a] [b])

Real Protocol | XZDH

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
Record that we are the initiator, $\mathcal{I}$
Retrieve PK_I and SK_I from $\bar{\mathcal{G}}^{XZDH}$
Retrieve $(\texttt{spk}, \mathcal{I}, pk, sk, \xi)$ from $\bar{\mathcal{G}}^{XZDH}$
Record PK_I, SK_I, pk, sid, and Φ
Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}()$
Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I \| pk \| \xi$ using aux to route

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
Record that we are the responder, $\mathcal{R}$
Retrieve PK_R and SK_R from $\bar{\mathcal{G}}^{XZDH}$
Retrieve PK_I from $\bar{\mathcal{G}}^{XZDH}$
Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I \| pk \| \xi)$ **to** $\mathcal{R}$:
if $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
if $(\neg(\text{SVerif}(PK_I, pk, \xi)))$ {
  Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
}
Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
Compute $\kappa = \text{KDF}_1((g^i)^r \| pk^r \| (PK_I)^r \| Q_k)$
Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
Let $t = \mathcal{I} \| \mathcal{R} \| g^i \| g^r \| PQ_I \| Q_R \| pk \| \Phi$
Compute $mac = \text{MAC}(M_k, t)$
Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
Erase r, Q_k, κ, and M_k
Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| mac \| \sigma$ to $\mathcal{I}$
Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| g^p \| Q_P \| mac \| \sigma)$ **to** $\mathcal{I}$:
Retrieve PK_P from $\bar{\mathcal{G}}^{XZDH}$
Retrieve $(\texttt{spk}, \mathcal{I}, pk', sk, \xi)$ from $\bar{\mathcal{G}}^{XZDH}$
if $(pk \neq pk')$ {
  Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
}
Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| pk \| \Phi$
if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t)))$ {
  Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
}
Compute $Q_k = \text{QRDecaps}(SQ_I, Q_P)$
Compute $\kappa = \text{KDF}_1((g^p)^i \| (g^p)^{sk} \| (g^p)^{SK_I} \| Q_k)$
Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
Compute $mac' = \text{MAC}(M_k, t)$
if $(mac \neq mac')$ {
  Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
}
Erase i, SQ_I, Q_k, κ, and M_k
Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 6.14 INCRIMINATION PROCEDURE FOR XZDH. (Ref: 157)

Subroutine | $\text{IncProc}_{\text{XZDH}}(sid, \bar{I}, \overline{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \bar{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I, g^\Gamma)$ **from** $\mathcal{S}$:
Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QREncaps}(PQ_I)$
Compute $\kappa = \text{KDF}_1((g^i)^r \| (g^\Gamma)^r \| (PK_I)^r \| Q_k)$
Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
Let $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^r \| PQ_I \| Q_R \| g^\Gamma \| \Phi$
Compute $mac = \text{MAC}(M_k, t)$
Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
Compute $\psi = \text{“}R\text{”} \| g^r \| Q_R \| mac \| \sigma$
Send $(\texttt{inc}, sid, \bar{I}, \overline{R}, \psi)$ to $\mathcal{S}$
Locally output k
Halt

There are a few differences between Theorem 6 and Theorem 5. The shared functionality, $\bar{\mathcal{G}}^{XZDH}$, enables the use of signed prekeys. The incrimination procedure shown in Algorithm 6.14, $\text{IncProc}_{\text{XZDH}}$, is identical to $\text{IncProc}_{\text{ZDH}}$ except for the addition of a signed prekey g^Γ as input, and the use of g^Γ to derive the shared key. Although $\mathcal{N}_{ZDH}$ serves as the non-information oracle, the XZDH simulator must provide it with signed prekeys so that the correct session keys are derived.

6.5.2.1 Forward Secrecy of XZDH

XZDH offers forward secrecy that exists somewhere between the traditional “strong” and “weak” definitions. In the context of Theorem 6, the property can now be expressed precisely: if a party $\overline{P}$ outputs a key κ and a partner identifier $\overline{P'}$, then $\mathcal{Z}$ can never distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^\lambda$ unless $\overline{P}$ or $\overline{P'}$ was corrupted before the corresponding session completed, or $\mathcal{S}$ aborted the session and $\overline{P'}$ did not subsequently issue an `spk-refresh` message to $\bar{\mathcal{G}}^{XZDH}$.

By definition, $\mathcal{N}_{ZDH}$ cannot assist $\mathcal{Z}$ with breaking forward secrecy. The only viable approach for $\mathcal{Z}$ is to abort the session and derive information from IncProc_{XZDH}, similarly to the attack on strong forward secrecy described in Section 6.5.1. The main difference with XZDH is that $\mathcal{Z}$ needs to compute $g^{\Gamma r}$ in order to provide the correct input to the KDF_1 random oracle. If $\bar{I}$ has not issued an `spk-refresh` message since the end of the session, then $\mathcal{Z}$ can send `spk-retrievesecret` from the corrupted $\bar{I}$ to derive $(g^r)^\Gamma$. However, if the signed prekey has been refreshed, then there is no way for $\mathcal{Z}$ to recover Γ from $\bar{\mathcal{G}}^{XZDH}$, thereby satisfying the forward secrecy property.

6.5.3 Proof of Theorem 6

(Sketch) The proof of Theorem 6 is nearly identical to the proof of Theorem 5 in Section 6.4. For this reason, the sketch only highlights the differences in simulator construction. Section 6.5.4 constructs the simulator and Section 6.5.5 demonstrates indistinguishability.

6.5.4 Simulator Construction

The simulator $\mathcal{S}$ behaves exactly as in Section 6.4.1 except that in the proof for Theorem 6, all references to $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ are replaced by references to $\bar{\mathcal{G}}^{XZDH}$.

6.5.4.1 Receipt of `solicit` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$

When $\mathcal{S}$ receives $(\texttt{solicit}, sid, \bar{I}, \Phi_I)$ from $\mathcal{F}^{+}_{1psp\text{-}keia}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ requests the current signed prekey for $\bar{I}$ by sending $(\texttt{spk-retrieve}, \bar{I})$ to $\bar{\mathcal{G}}^{XZDH}$, and receiving $(\texttt{spk}, \bar{I}, g^{\Gamma}, \xi)$ in response. $\mathcal{S}$ computes $\psi_1 = \text{“}I\text{”} \parallel g^i \parallel PQ_I \parallel g^{\Gamma} \parallel \xi$ using the g^i and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ also records the value Φ_1 for later reference.

6.5.4.2 Receipt of `establish` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$

This case is mostly the same as the case from Section 6.4.1, with the exceptions noted below.

When checking the validity of ψ_1, $\mathcal{S}$ also examines the signed prekey g^{Γ} with signature ξ from ψ_1. $\mathcal{S}$ retrieves $PK_I = g^I$, the long-term public key for $\bar{I}$, from $\bar{\mathcal{G}}^{XZDH}$ using a `retrieve` message. If $\text{SVerif}(g^I, g^{\Gamma}, \xi) \neq \text{TRUE}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, delivers the resulting `abort` to $\bar{R}$ immediately, and withholds the `abort` message to $\bar{I}$.

$\mathcal{S}$ also acts slightly differently when constructing ψ_2:

- If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message $\psi_1{}'$, but $\psi_1 \neq \psi_1{}'$, then $\mathcal{S}$ behaves as in Section 6.4.1 with the exception of passing g^{Γ} to IncProc as part of the `inc` message. IncProc returns a properly constructed ψ_2 message for XZDH.
- Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a forged proof σ_R with tag $t = \text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^r\|PQ_I\|Q_R\|g^{\Gamma}\|\Phi_R$. The method for choosing the ephemeral keys (g^r and Q_R) and the keys used to forge the proof depends on the environment:

- If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ behaves as in Section 6.4.1, except that it sends $(\texttt{authenticate}, \{g^\Gamma, g^I\}, t)$ to $\mathcal{N}$ when requesting *mac*.
- If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I, \{g^\Gamma, g^I\})$ to $\mathcal{N}$. $\mathcal{S}$ proceeds to acquire r, $SK_I = I$, and σ_R as in Section 6.4.1. $\mathcal{S}$ computes $\kappa = \text{KDF}_1((g^i)^r \| (g^\Gamma)^r \| (g^I)^r \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and then $mac = \text{MAC}(M_k, t)$.

6.5.4.3 Receipt of ψ_2 by uncorrupted $I^{(s)}$

This case is mostly the same as the case from Section 6.4.1, with two differences. Firstly, when checking that ψ_2 matches the ψ_1 message sent by $I^{(s)}$, $\mathcal{S}$ first retrieves the latest signed prekey for $\bar{I}$ by sending $(\texttt{spk-retrieve}, \bar{I})$ to $\bar{\mathcal{G}}^{XZDH}$, and receiving $(\texttt{spk}, \bar{I}, g^{\Gamma'}, \xi)$ in response. If g^Γ was the signed prekey transmitted in ψ_1 and $g^\Gamma \neq g^{\Gamma'}$, then $\mathcal{S}$ treats ψ_2 as invalid and aborts $\bar{I}$. Secondly, when checking the validity of mac_P, $\mathcal{S}$ sends $(\texttt{verify}, \{g^\Gamma, g^I\}, t, mac_P)$ to $\mathcal{N}$, where tag $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^p \| PQ_I \| Q_P \| g^\Gamma \| \Phi_I$.

6.5.5 Proof of Indistinguishability

The proof of indistinguishability given in Section 6.4.2 also applies here with several additional remarks.

The session keys produced by IncProc_{XZDH} and $\mathcal{N}_{ZDH}$ continue to be indistinguishable to $\mathcal{Z}$ due to the CDH assumption for $\mathbb{G}$, since the inputs to KDF_1 still include terms that $\mathcal{Z}$ cannot compute; the inclusion of $g^{\Gamma r}$ does not negate the computational independence of the random oracle's outputs.

Although ψ_1 now contains long-term information—g^Γ and its signature ξ—these values do not help $\mathcal{Z}$ to distinguish between $\mathcal{S}$ and $\mathcal{A}$. $\mathcal{S}$ ensures that $R^{(s)}$ checks the validity of the signature; if ξ is invalid, $\mathcal{S}$ aborts the session in the same manner as a real responder. If $\mathcal{Z}$ replays a signed prekey or generates one from a corrupted $\bar{I}$ via $\bar{\mathcal{G}}^{XZDH}$, $\mathcal{S}$ still effectively simulates $R^{(s)}$ honestly. Replaying the signed prekey or sending a corrupted one also does not affect the security properties of the MAC or proof in ψ_2.

Finally, $\mathcal{S}$ also aborts $\bar{I}$ when ψ_2 includes an outdated g^Γ value (i.e., $\bar{I}$ has refreshed the signed prekey stored by $\bar{\mathcal{G}}^{XZDH}$ after $\mathcal{S}$ simulated ψ_1). This is precisely the behavior of a real initiator, who always knows which signed prekey is being distributed and who refuses to accept responses using an old key. ■

6.6 Chapter Summary

This chapter sketched security proofs for the new DAKEs introduced in Chapter 5. The DAKEs and their security properties were formally defined in the GUC framework, and shown to be indistinguishable from new ideal functionalities. These ideal functionalities were shown to provide the desired security properties, thereby completing the simulation-based security proofs. This proof technique provides good evidence that the DAKEs are secure even when embedded in a higher-level protocol and executed concurrently.

Section 6.1 surveyed techniques for proving the security of strongly deniable DAKEs, justified the use of the GUC framework for the proof sketches in this chapter, and defined a shared functionality that captures the cross-session public keys used by the new DAKEs (Algorithm 6.1). Section 6.2 sketched the security proof for DAKEZ in the GUC framework, which includes the definition of a new ideal functionality (Algorithm 6.2) capturing contributiveness. Section 6.3 sketched the security proof for Spawn$^+$ and defined a new ideal functionality (Algorithm 6.6) that captures the behavior of Spawn$^+$, ZDH, and XZDH. Section 6.4 sketched the security proof for ZDH as an extension of the proof for Spawn$^+$. Finally, Section 6.5 sketched the security proof for XZDH, including a formal specification in the GUC framework of the partial forward secrecy provided by signed prekeys.

The two chapters in Part II defined new strongly deniable DAKEs and established that they are secure. While these DAKEs can be used to eliminate deniability related insider attacks, they are fundamentally limited to two-party settings. Because DAKEZ and XZDH can serve as drop-in replacements for 3DH and X3DH, they can be used in pairwise secure group messaging protocols like the ones mentioned in Section 2.8. Unfortunately, this approach lacks the scalability of secure group messaging protocols built with true GKEs, as discussed in Chapter 3. Part III bridges the gap by developing a new CGKA-like scheme. This new scheme uses some of the same high-level techniques to achieve strong deniability as the DAKEs in this chapter (in particular, a variant of the proof statements in the SoKs), but in a group context.

PART III

Secure Group Messaging

CHAPTER 7 | Designing A Secure Group Messaging Protocol

In this chapter: Protocol

Current proposals for secure group messaging protocols, like MLS, do not provide robust protection against insider attacks. These attacks grow increasingly important as the size of the group increases, because there are more insiders that can potentially act maliciously. Since the focus of recent research has been to improve the scalability of protocols, defending against these attacks has become more important than ever before. This part of the dissertation introduces a new protocol called Safehouse to address this shortcoming.

> This dissertation is a static document and may not represent the current state of the Safehouse protocol. The most recent protocol definition, security notices, news, and implementations will be made available at safehouse.im. Visit the website for the most recent information.

Safehouse is a cryptographic protocol that extends the notion of a DGKE in a similar manner to CGKAs,[1] providing key establishment and management over time for a dynamic group. Safehouse can be used to easily construct secure group messaging protocols with insider security for a wide range of applications. As discussed in Section 2.6, secure messaging protocols are composed of three mostly orthogonal design layers: trust establishment (see Section 2.6.1) enables the secure binding of long-term keys to identities, conversation security (see Section 2.6.2) uses the long-term keys to protect the actual messages in a manner that provides the desired security properties, and private transport (see Section 2.6.3) protects the metadata. Safehouse is a protocol located entirely in the conversation security layer; it is compatible with any appropriate trust establishment and private transport implementations that make sense for the target application. Safehouse leaves room for the protocol designer to choose important aspects of the conversation security layer. For example, a secure group messaging protocol constructed using Safehouse

[1] See Chapter 3 for an overview of GKEs and DGKEs, and Chapter 4 for a discussion of CGKAs. Differences between Safehouse and CGKAs are discussed later in this chapter.

retains full control over how messages are encrypted—Safehouse provides a mechanism to derive domain-separated shared secrets that can be used as key material, but does not impose any particular cryptosystems.

Secure messaging protocols can be specified at three general levels of abstraction, as defined in Section 2.2: an abstract protocol defines the high-level data flows between participants, a wire protocol defines binary-level network transmissions, and a tool is a concrete software implementation of the protocol. Safehouse is defined herein as an abstract protocol. Generally speaking, it is best to specify secure messaging protocols as wire protocols as part of standardization projects. Safehouse cannot be entirely expressed as a tool because it supports scenarios with many different network models; a secure messaging tool may include a particular instantiation of Safehouse in combination with trust establishment and private transport schemes, but the Safehouse protocol itself is more general.

New secure messaging protocols should be driven by the needs of users in the context of the overall messaging tool environment (including popular insecure messengers). Section 2.1 surveyed notable user studies that can help to inform protocol design beyond what is simply academically interesting or valuable to the protocol designer. Ideally, a new secure group messaging protocol would be widely applicable to a variety of currently popular communication paradigms in order to maximize its usefulness (assuming that wide coverage is possible without compromising security). Given all of these considerations, Safehouse is designed to provide security for the following popular use cases, among others:

- A small group of low-risk friends converses using text messaging from mobile devices.
- A medium-sized group of low-risk friends and acquaintances uses mobile and desktop devices to communicate in an invite-only Slack channel.
- A medium or large group of low-risk users discuss a specific topic in an IRC channel that anyone can join.
- A small group of low-risk friends talk in a Zoom video call using mobile and desktop devices.
- A small- or medium-sized group of users from several businesses host a private and confidential video conference discussing a potential deal.
- A small group of political campaign staff discuss strategy in a private text-based channel.
- A small group of high-risk users conduct a private and pseudonymous text- or audio-based conversation within a repressive regime.

- A medium or large group of group of medical professionals receives text-based alerts from a hospital's group communication system.
- Someone accused of wrongdoing defends themselves during a video call with a small group.

The terms "low-risk" and "high-risk" in this list refer to the definitions by Halpin et al. [HEM18], as discussed in Section 2.1. The terms "small", "medium", and "large" in this list are very loosely defined to mean approximately 10, 100, and 1000, respectively. These group sizes generally tend to be associated with messaging tools that have very different user experiences.

The list of target use cases above is not comprehensive, but it serves as a good test for evaluating the appropriateness of design decisions. Each of the common target scenarios has a distinct set of security requirements that can be informed by the user studies in Section 2.1. In some cases, users should have identities that are globally constant, while in others, each message should be anonymous. In some cases, sending messages and participating in the group should be deniable, while in others, messages should be non-repudiable. In some cases, conversation transcripts should be globally consistent, while in others, messages can be randomly delivered to a subset of participants without significantly impacting utility. Closer inspection reveals even more differences between the desired properties for the scenarios. Surprisingly, Safehouse is able to efficiently provide security in all of these settings by providing a few configurable behaviors. One notable scenario that is not included in the list is the case of a large group of high-risk activists using a text-based group conversation to coordinate a protest. This scenario requires very different security properties than the others and, although Safehouse can be used to implement a tool for this application, it is best served by a separate protocol.[2]

Informally, Safehouse exhibits the following properties:

- **Group key exchange**: All members of the group gain knowledge of a shared symmetric secret key called the group key. The group key is internal to the protocol. Safehouse provides a function to derive shared application-specific keys from the group key. Messages can be encrypted using one of these derived keys.
- **Dynamic group membership**: Over time, members can join and leave the group. The group key is replaced after every membership change; a message encrypted using the group key

[2] ^ A protocol for this purpose should provide insider security, deniability, an invitation system, no long-term keys, and a frequently rotated shared key used to encrypt and authenticate all messages anonymously within the group. Many of the features provided by Safehouse are unnecessary for this application. Additionally, Safehouse manages many keys that are useful for the other applications, but not for the protest coordination scenario; this adds unnecessary overhead. A more efficient protocol for this scenario could likely be developed using the cryptographic primitive introduced in Chapter 8; this is left to future work.

can only be read by the current members of the group. No colluding group of outsiders, not even former group members that have left the group, can break message confidentiality.

- **Mass membership changes**: An external group of online participants that can communicate interactively are able to collectively join the group in a single *mass join* operation that is more efficient than individual joins. Likewise, a subgroup of group members can be evicted in one *mass evict* operation more efficiently than evicting them individually.

- **Forward Secrecy**: Members can send a special type of message that replaces the group key with a new group key. Once another group member processes this special message, compromising that member's long-term and ephemeral keys is insufficient to decrypt messages encrypted using previous group keys.

- **(Optional) Post-Compromise Security**: Consider an adversary that passively records network communication and also compromises a snapshot of the long-term and ephemeral keys of a subgroup. After every compromised group member has sent a special type of message, the adversary is no longer able to passively decrypt new messages. In effect, Safehouse is "self-healing" in this mode. This protection is optional because it comes at an additional performance cost that is not always desirable.

- **Semi-trusted server**: A semi-trusted always-online central server stores and forwards protocol messages to all members of the group. The server is responsible for enforcing a total order on protocol messages. The server can break availability by refusing to relay data. The server can also fragment the group by relaying different legitimately created messages to different subgroups, but these subgroups will no longer accept messages from each other; this is called a *fork*. These operations are not considered attacks.

- **Non-Interactivity**: Group membership changes and other operations can be performed by a single group member interacting with the central server, even while all other group members are offline. When a group member comes back online, they receive a list of every message since they disconnected. This list is sufficient to recover the interim conversation and the current group key, even if all other group members are offline.

- **Publicly verifiable operations**: The central server is able to verify the correctness of all operations performed by group members, even in cases where the server is prevented from decrypting data. The server is able to view a carefully selected subset of the group state, allowing it to enforce access control policies and implement spam prevention.

- **Insider security**: Malicious group members are not able to sabotage the group, even by colluding with each other and with the central server. Specifically, these adversaries cannot

cause honest group members to communicate while possessing differing group states. At most, the central server (without any insider assistance) can permanently fork the group, preventing future communication between subgroups with different states.

- **(Optional) Sender authentication**: Safehouse generates individual ephemeral signing keys for each group member. These keys can be used by a secure group messaging protocol to sign messages, thereby authenticating the sender to other group members. A messaging protocol may also choose to hide this authorship from outsiders by encrypting the signature using the group key.

- **(Optional) Participant authentication**: When configured to do so, Safehouse explicitly authenticates the long-term identities of all group participants. When combined with a trust establishment mechanism, this allows users to verify that they are communicating with each other. When this feature is disabled, participants are identified solely by their ephemeral signing keys.

- **(Optional) Strong deniability**: Groups may optionally provide offline and online participation deniability, as discussed in Section 2.4. The group can be configured to provide non-repudiation, offline deniability only, or strong deniability.

- **Invitations**: The group requires an invitation in order to join. An inviter generates secret key material (the invitation) that is delivered to an invitee out of band and subsequently used to join the group. There are two types of invitations: a *targeted invitation* can only be used by an invitee with a designated long-term public key, while a *bearer invitation* can be used by any invitee. Invitations may have expiration dates, and bearer invitations may have a limit on the number of times that they may be used. Invitation policies are cryptographically enforced. All group members are able to see a list of currently outstanding invitations and, if they were in the group at the time when the invitation was sent, which group member issued them. A secure group messaging protocol built with Safehouse can implement an "open invitation" group by publishing an unlimited-use bearer invitation.

- **Anonymity preservation**: Safehouse never leaks long-term public keys to the network or to the central server. When invitations are used, it is not possible for the adversary to determine the long-term identity of the inviter or invitee, but the validity of the invitation used to join a group is nevertheless publicly verifiable. Note that these protections are necessary but usually insufficient to provide anonymity in practice—a member's identity may still be inferred from network details like an IP address. However, if Safehouse is combined with an underlying network anonymity layer, this property ensures that the system does not undermine any anonymity guarantees provided by that layer. While Safehouse preserves the anonymity of

the members' identities, it does not attempt to hide group activity from the semi-trusted server; the server is able to observe how the (anonymous) members alter the group state over time. In particular, the server can observe when members are added to or removed from the group, and it can maintain a history of all actions performed by each member. However, anonymity preservation prevents the server from linking these histories to long-term identities in the absence of external knowledge (e.g., a social graph obtained from another service).

- **Secure property storage**: Safehouse supports the storage of an arbitrary key-value table in the group state. Keys are always public, but values may optionally be encrypted. Encrypted values can only be read by group members. New group members can decrypt existing values immediately when joining the group, without waiting for other group members to come online. This property storage can be used by developers to implement high-level messaging protocol features that are expected in modern tools, such as channel topics, pinned messages, and user settings.

Safehouse most closely resembles a CGKA like TTKEM using the insider security extensions proposed by Alwen et al. [ACJM20], as discussed in Chapter 4. However, Safehouse takes a different path that differentiates it from CGKAs. CGKAs like TreeKEM allow a participant to non-interactively initialize a group and immediately send messages, even without the consent of the recipients. This is similar to how email operates. Safehouse rejects this functionality, requiring explicit consent (through protocol participation) from members in order to join a group, as recommended by Dev et al. [DMC20] as a result of their user study. This also improves the security of the scheme, because it enables a guarantee that participants always have explicit authentication for all recipients that could possibly decrypt a sent message. Unlike the CGKA proposed by Alwen et al. [ACJM20], Safehouse's configurable insider security mechanism does not rely on expensive zero-knowledge proofs about cryptographic hash functions. Another important difference from a scheme like MLS is that Safehouse gives the central server verifiable access to certain public information about the group state, allowing it to enforce access control policies; a malicious server that fails to do so will at most succeed at forking the group, which is not considered an attack. In contrast, MLS treats the server as a blind routing mechanism[3] in order to improve performance. Safehouse also provides more features than MLS, which strives to be a "bare minimum" specification in order to encourage standardization. Safehouse takes the view that functionality like group invitations and secure property storage, which are desirable in many of the target applications, require implementations that are tightly coupled to the underlying cryptographic machinery and thus should be included in the protocol definition. The configurable aspects of Safehouse are normally established by a secure messaging protocol designer and

[3] Despite this, as discussed in Section 4.3, MLS encourages that the server applies a total order to routed messages to maximize security.

hardwired into tools that implement the protocol. However, designers can also allow end users to configure Safehouse parameters by attaching a custom "mode" to the public group state, and then using this mode to determine how Safehouse should behave; Safehouse provides a location for this mode data in the group state and ensures that it is authenticated to prevent tampering by the central server.

Safehouse is a very complex protocol with many interacting components. This part of the dissertation introduces the protocol in a "bottom-up" approach, starting with the two largest and most complex sub-protocols. Chapter 8 introduces a new cryptographic primitive that is the main mechanism for achieving Safehouse's insider security guarantees. This new primitive is a contribution of independent interest: its applications reach beyond secure messaging. Chapter 9 introduces a new GKE that can be used to efficiently establish shared secrets for a given key tree. This approach enables highly efficient mass join operations in Safehouse. It is also directly applicable to other tree-based CGKAs like TreeKEM. Chapter 10 introduces the Safehouse protocol itself. Finally, because Safehouse is applicable to a wide range of applications, Chapter 11 describes a specific sample secure group messaging tool implemented using Safehouse in order to illustrate the practicality of the system.

CHAPTER 8 | BRAKEM: Publicly Verifiable Key Encapsulation

In this chapter: Protocol | Background | Implementation | Proof | Evaluation

One of the main challenges in designing modern secure group messaging protocols is that participating devices are expected to go offline frequently and without warning. It is no longer reasonable to expect that all participants can always maintain active connections, as in the case of an IRC server for users with desktop computers and broadband Internet connections; the advent of mobile devices like laptops and smartphones that hop between networks and are often powered off has changed the messaging landscape. To support these use cases, secure group messaging protocols must allow participants to send messages and perform operations non-interactively, since there may never be a time when more than one group member is online simultaneously. Chapter 4 discussed how the ART and TreeKEM protocols approach this problem. In both cases, the protocols include situations where the participant sending a message generates a new public key for another group member. The message that they transmit installs this new public key in the shared group state, and also includes enough information for the other group member to compute the new private key. This is the fundamental step that allows these protocols to implement non-interactive operations.

At its core, Safehouse also employs a key encapsulation technique. However, unlike ART and TreeKEM, Safehouse aims to provide insider security. An important step toward this goal is allowing outsiders (specifically, the semi-trusted central server) to publicly verify that newly transmitted public keys were formed correctly. Section 8.1 formally defines the exact primitive that achieves these goals. This primitive can be instantiated with many different zero-knowledge proof systems, each offering different performance and security tradeoffs. This chapter discusses a few specific instantiations. Section 8.2 presents a construction of the new primitive using a traditional "Schnorr proof" system. Two options are given: a high-performance construction with an unusual hardness assumption, and a slower variant that relies on only very conservative assumptions. Section 8.3 describes important factors for practical deployments, and Section 8.4 provides security proofs for the construction. Section 8.5 describes how the primitive could be constructed using one of the more recent zero-knowledge proof systems. This construction has significantly different performance characteristics that are advantageous in certain settings, but it

requires stronger security assumptions. Section 8.6 discusses how to select a construction based on practical considerations.

8.1 Batch Recursive Attested Key Encapsulation Mechanism

To motivate the new cryptographic primitive presented in this section, consider this scenario:

> Two participants, $\mathcal{U}_1$ and $\mathcal{U}_2$, each have an asymmetric key pair: $\mathcal{U}_1$ knows (pk_1, sk_1) and $\mathcal{U}_2$ knows (pk_2, sk_2). $\mathcal{U}_1$ and $\mathcal{U}_2$ have previously securely shared their public keys and have performed trust establishment. Another asymmetric key pair (pk, sk) is known to both participants (in particular, they both know the private key sk). A simple two-party secure messaging scheme can be implemented by encrypting messages to pk, enabling both parties to decrypt the messages using sk. Suppose that sk has been compromised and the participants want to replace (pk, sk) with a new shared key pair (pk^*, sk^*) to achieve post-compromise security. Moreover, they wish to accomplish this non-interactively (i.e., while the operation is performed by participant $\mathcal{U}_i$, participant $\mathcal{U}_{3-i}$ may be offline). The participant performing the update must non-interactively generate a transmission that can be used by the other participant at a later time to derive the new key pair, while preserving key secrecy against passive network adversaries.

This scenario is similar to the problem that is solved by DH ratchets in protocols like OTR. However, it is slightly unusual in that the shared secret is from an asymmetric cryptosystem, rather than a typical symmetric shared key. The purpose of this deviation is to assist with achieving insider security, as will soon become clear.

A first attempt to implement the protocol described in the scenario above might be to use an asymmetric KEM scheme consisting of the following functions:

- **KEM.KeyGen**$(s) \to (pk, sk)$: Takes as input randomness s, then outputs a public key pk with corresponding private key sk.
- **KEM.Encapsulate**$(pk; s') \to (pk^*, sk^*, C)$: Takes as input a public key pk and randomness s', then outputs a new key pair (pk^*, sk^*) and a ciphertext C.

- **KEM.Decapsulate**$(pk, sk, pk^*, C) \rightarrow (sk^*)$: Takes as input a key pair (pk, sk), a new public key pk^*, and a ciphertext C. Outputs either a private key sk^* corresponding to pk^* or the special symbol $\perp$ denoting an error.

This definition of an asymmetric KEM assumes that it is possible to publicly verify that a public key is *valid*. A public key pk is valid if there exists some randomness s such that KEM.KeyGen$(s) \rightarrow (pk, sk')$ for some sk'.[1] Moreover, the KEM definition assumes that every valid public key *corresponds* to exactly one private key: for any public key pk, there exists a unique corresponding private key sk such that for any randomness s with KEM.KeyGen$(s) \rightarrow (pk, sk')$, $sk' = sk$. A scheme satisfying these properties (the ability to validate public keys and a unique correspondence between public and private keys) is called *admissible*, and this chapter restricts its attention to such schemes. Informally, a secure instantiation of an asymmetric key encapsulation would provide correctness and key secrecy. Correctness ensures that if KEM.Encapsulate is called for a valid pk and KEM.Decapsulate is called on the resulting ciphertext with the expected keys, then both parties will recover sk^*. Key secrecy ensures that it is only possible to learn sk^* if one knows sk or was the caller of KEM.Encapsulate. For ease of presentation, all schemes in this chapter are implicitly parameterized by a security parameter λ that influences the size of all values. As a notational convenience, this chapter usually omits the randomness parameter in function calls in order to denote that a random value of the appropriate length is implicitly sampled.

An asymmetric KEM can be used to solve the aforementioned problem as follows: when $\mathcal{U}_i$ wants to update the shared key pair, it calls KEM.Encapsulate(pk_{3-i}) to generate the new key pair (pk^*, sk^*), then sends C to $\mathcal{U}_{3-i}$. If all goes well, when the other participant comes back online and receives the message, it can call KEM.Decapsulate to recover sk^*.

The solution above allows two participants to non-interactively update a shared asymmetric key pair. Next, consider a more difficult version of the problem: instead of (pk_1, sk_1) and (pk_2, sk_2) corresponding to individual participants, these key pairs themselves are each shared by a subgroup of participants. In other words, each participant in the group either knows sk and sk_1, or it knows sk and sk_2. Replacing (pk, sk) with a new key pair (pk^*, sk^*) now requires encapsulating sk^* to every participant in the group. Luckily, since each participant knows either sk_1 or sk_2, sk^* only needs to be encapsulated to these two keys. The asymmetric KEM scheme above can be modified for this application, resulting in what might be called an (asymmetric) "dual KEM" (DKEM):

- **DKEM.KeyGen**$(s) \rightarrow (pk, sk)$: Takes as input randomness s, then outputs a public key pk with corresponding private key sk.

[1] Of course, actually *finding* s given only pk should be computationally infeasible in order for the scheme to be secure.

- **DKEM.Encapsulate**$(pk_1, pk_2; s') \rightarrow (pk^*, sk^*, C_1, C_2)$: Takes as input public keys pk_1 and pk_2, and randomness s', then outputs a new key pair (pk^*, sk^*) and ciphertexts C_1 and C_2.
- **DKEM.Decapsulate**$(i, pk_1, pk_2, sk, pk^*, C) \rightarrow (sk^*)$: Takes as input an index $i \in \{1, 2\}$, public keys pk_1 and pk_2, a private key sk corresponding to pk_i, a new public key pk^*, and a ciphertext C. Outputs either a private key sk^* corresponding to pk^* or the special symbol $\perp$ denoting an error.

Assume that pk_1 and pk_2 have been generated by DKEM.KeyGen and both sk_1 and sk_2 have somehow been shared among the appropriate participants in the two subgroups. A participant can call DKEM.Encapsulate(pk_1, pk_2) to generate a replacement key pair (pk^*, sk^*). C_i is sent to participants in the subgroup corresponding to pk_i for $i \in \{1, 2\}$. Each participant in the group can now call DKEM.Decapsulate with the appropriate parameters for their subgroup to recover sk^*, completing the non-interactive key update.

The problem solved by DKEM above intentionally omits an important detail: how do the two subgroups establish shared access to sk_1 and sk_2 in the first place? This problem can be solved recursively. As a base case, two parties can establish a shared key using a simple asymmetric KEM, resulting in a shared key pair for a subgroup of size two. As the recursive step, two subgroups can establish a shared key by using the DKEM scheme, resulting in a shared key pair known by the union of the subgroups. This approach is recognizable as a method to non-interactively establish a key tree of the type discussed in Section 3.5. However, there is one important caveat: this approach only works if the key pairs output by the key generation and encapsulation functions come from the same key space. The standard definition of an asymmetric KEM does not necessarily provide this property. In this chapter, KEMs that share a key space between key generation and encapsulation are called *recursive*. When a scheme satisfies the DKEM definition given above and is also recursive, it can be called a "dual recursive KEM" (DRKEM).

One additional variation is required to meet the minimum standards for a KEM that can be used by Safehouse: the scheme must provide insider security. In this context, that means that the results of the key encapsulation must be publicly verifiable. The scheme must convince any observer, whether part of the group or not, that all recipient participants will be able to compute the new private key. Without this verifiability, a malicious participant that calls the key encapsulation function for a DRKEM scheme could perform attacks like sending C_1 to the first subgroup as normal, but sending a corrupted value C_2' to the second subgroup; this attack would prevent participants in the second subgroup from recovering sk^*, without any honest participants in the first subgroup noticing a problem. To prevent these attacks, the scheme can be modified to include verification functions that test the correctness of a key encapsulation. In practice, these functions can be implemented with NIZKPKs. Modifying the DRKEM definition in this way produces a "dual recursive attested KEM" (DRAKEM):

- **DRAKEM.KeyGen**$(s) \rightarrow (pk, sk)$: Takes as input randomness s, then outputs a public key pk with corresponding private key sk.
- **DRAKEM.Encapsulate**$(pk_1, pk_2; s') \rightarrow (pk^*, sk^*, C_1, C_2, \pi)$: Takes as input public keys pk_1 and pk_2, and randomness s', then outputs a new key pair (pk^*, sk^*), two ciphertexts C_1 and C_2, and a correctness proof π.
- **DRAKEM.Decapsulate**$(i, sk, pk_1, pk_2, pk^*, C_1, C_2, \pi) \rightarrow (sk^*)$: Takes as input an index $i \in \{1, 2\}$, public keys pk_1 and pk_2, a private key sk corresponding to pk_i, a new public key pk^*, ciphertexts C_1 and C_2, and a correctness proof π. Outputs either a private key sk^* corresponding to pk^* or the special symbol $\perp$ denoting an error.
- **DRAKEM.Verify**$(pk_1, pk_2, pk^*, C_1, C_2, \pi) \rightarrow b$: Takes as input public keys pk_1 and pk_2, a new public key pk^*, ciphertexts C_1 and C_2, and a correctness proof π. Outputs a bit $b \in \{0, 1\}$.

DRAKEM differs from DRKEM by outputting a correctness proof π from DRAKEM.Encapsulate. This proof must be provided to DRAKEM.Decapsulate. The decapsulation function will fail and output $\perp$ with overwhelming probability if the proof is invalid. Otherwise, participants are assured that any participant in either subgroup will be able to recover sk^*. This property is called *verifiability*. Although only one ciphertext is needed to recover the key, both ciphertexts are provided to the decapsulation function so that it can verify the proof. Additionally, a new function called DRAKEM.Verify is added to the scheme. This new function allows an outsider to verify that the key encapsulation was performed correctly, even though the new function does not require any secret inputs. If the function outputs $b = 1$, then the caller is assured with overwhelming probability that all participants in the group will be able to recover sk^*. This property is called *public verifiability*.

While it is possible to build Safehouse using DRAKEM, an extra tweak to the scheme admits constructions that are far more efficient in practice. Instead of encapsulating one key to two subgroups, the scheme can be modified to support encapsulating m different keys to independent subgroups of arbitrary size. This tweak allows constructions to potentially use more efficient "batching" techniques than could be achieved by repeated application of DRAKEM. The resulting scheme is called Batch Recursive Attested Key Encapsulation Mechanism (BRAKEM). The complete scheme is as follows:

- **BRAKEM.KeyGen**$(s) \rightarrow (pk, sk)$: Takes as input randomness s, then outputs a public key pk with corresponding private key sk.

- **BRAKEM.Encapsulate**$(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$: Takes as input m sets of public keys $R_1, \ldots, R_m$ and randomness s', then outputs m new key pairs (pk_i^*, sk_i^*) for $1 \leq i \leq m$, ciphertexts $C_1, \ldots, C_m$, and a correctness proof π.
- **BRAKEM.Decapsulate**$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow (sk_i^*)$: Takes as input m sets of public keys $R_1, \ldots, R_m$, indices $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$ identifying a public key in one of the sets, a private key sk corresponding to the identified public key, m new public keys $pk_1^*, \ldots, pk_m^*$, ciphertexts $C_1, \ldots, C_m$, and a correctness proof π. Outputs either a private key sk_i^* corresponding to pk_i^* or $\perp$.
- **BRAKEM.Verify**$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow b$: Takes as input m sets of public keys R_i, new public keys pk_i^*, and ciphertexts C_i for $1 \leq i \leq m$, as well as a correctness proof π. Outputs a bit $b \in \{0, 1\}$.

The BRAKEM scheme definition is applicable in contexts beyond secure group messaging, and its self-contained nature makes it easier to reason about its security. Safehouse uses a slightly modified version of BRAKEM that is more convenient to use in non-interactive secure group messaging protocols, but the algorithmic steps remain the same as in the corresponding BRAKEM construction. This chapter focuses on BRAKEM constructions.

When defining the security properties for BRAKEM, a set of public keys R_i (called a *ring*) is *valid* if and only if every public key $pk_{i,j} \in R_i$ is valid, as per the earlier definition of admissible KEMs. $R_{i,j}$ denotes the j^{th} public key in R_i. The variables in each security definition are independent of other definitions. A secure BRAKEM instantiation has the following properties:

- **Correctness**: if an honest sender encapsulates keys, then each of the intended recipients can recover the expected private key, and public observers will accept the encapsulation as valid. If:
  - every R_i for $1 \leq i \leq m$ is valid; and
  - BRAKEM.Encapsulate$(R_1, \ldots, R_m) \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$;

  Then:
  - every pk_i^* for $1 \leq i \leq m$ is valid;
  - BRAKEM.Decapsulate$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$ for any $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$ such that sk is the private key corresponding to $R_{i,j}$; and
  - BRAKEM.Verify$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow 1$.

- **Verifiability**: if an honest recipient successfully decapsulates a key, then it is assured that the ciphertext and proof could have been produced by an honest sender performing the encapsulation as intended.
  If:
  - every R_i for $1 \leq i \leq m$ is valid; and
  - BRAKEM.Decapsulate$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$ for some $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$ such that sk is the private key corresponding to $R_{i,j}$;

  Then:
  - every pk_i^* for $1 \leq i \leq m$ is valid; and
  - with overwhelming probability there exists an s' such that BRAKEM.Encapsulate$(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$.

- **Public Verifiability**: if an observer successfully verifies an encapsulation, then they are assured that all of the intended recipients are able to decapsulate the expected keys.
  If:
  - BRAKEM.Verify$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow 1$;

  Then:
  - R_i and pk_i^* for all $1 \leq i \leq m$ are valid; and
  - BRAKEM.Decapsulate$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$ for any $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$ such that sk is the private key corresponding to $R_{i,j}$.

- **Key Secrecy**: Any PPT adversary has negligible advantage in the following game. The challenger maintains a "challenge table" of key pairs, a "derivation table" that stores Boolean logic statements about how private keys can be derived, a "corruption log" with Boolean logic statements about corrupted keys, and a testing case initially set to $\perp$.
  1. The adversary is given access to an oracle $\mathcal{O}_{\text{KeyGen}}$. When called, the oracle computes BRAKEM.KeyGen$() \rightarrow (pk, sk)$, records (pk, sk) in the challenge table, and returns pk.
  2. The adversary is given access to an oracle $\mathcal{O}_{\text{Encapsulate}}(R_1, \ldots, R_m)$ that computes BRAKEM.Encapsulate$(R_1, \ldots, R_m) \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$. For each $1 \leq i \leq m$, the oracle checks to see if an entry $(R_{i,j}, sk_{i,j})$ exists in the challenge table for each $1 \leq j \leq |R_i|$. For a given $1 \leq i \leq m$, if there exists a value $1 \leq j \leq |R_i|$ such that no entry $(R_{i,j}, sk_{i,j})$ exists in the challenge table, then the oracle records

$\text{Corrupt}_{pk_i^*}$ = TRUE in the corruption log. For a given $1 \leq i \leq m$, if for all values $1 \leq j \leq |R_i|$ there exists an entry $(R_{i,j}, sk_{i,j})$ in the challenge table, then the oracle records (pk_i^*, sk_i^*) in the challenge table and, for all $1 \leq j \leq |R_i|$, it records the statement $\text{Corrupt}_{R_{i,j}} \rightarrow \text{Corrupt}_{pk_i^*}$ in the derivation table. In any case, the oracle returns $(pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$.

3. The adversary is given access to an oracle $O_{\text{Corrupt}}(pk)$. When called, $O_{\text{Corrupt}}(pk)$ checks to see if an entry (pk, sk) exists in the challenge table. If no entry exists, the oracle returns $\perp$. Otherwise, it adds Corrupt_{pk} = TRUE to the corruption log and returns sk.
4. The adversary outputs a key pair (pk, sk). If an entry (pk, sk) exists in the challenge table and it is not possible to conclude that Corrupt_{pk} = TRUE using the truth tables in the corruption log and the logical implications in the derivation table, then the adversary wins.

The definition of verifiability for BRAKEM is stronger than typical definitions for similar schemes, since it requires that any valid proof must actually be producible by the encapsulation function. Typically, "verifiability" would only require that decryption yields the same result for the other recipients,[2] which is a weaker requirement. BRAKEM uses the stronger definition because it is easily provable for the constructions presented in this chapter and it provides better security. As an added benefit, the definition is also simpler.

The BRAKEM key secrecy game is similar to the widely used extended Canetti-Krawczyk (eCK) security model for authenticated key exchanges [LLM07] with some modifications. Aside from the fact that BRAKEM supports batching encryptions to multiple rings simultaneously, the most significant difference with eCK is that the BRAKEM security definition does not include authentication, which greatly simplifies the modeling. Rather than defining key secrecy in terms of Turing machines with protocol sessions and session matching rules, the game-based definition implicitly represents communicating parties using the O_{KeyGen} and $O_{\text{Encapsulate}}$ oracles. By calling these oracles, the adversary can mimic the behavior of an active network adversary that can invoke and manipulate multiple parallel BRAKEM protocol sessions. $O_{\text{Encapsulate}}$ allows the adversary to provide adversarially generated keys in the recipient rings, but any such BRAKEM messages are assumed to be corrupted; nonetheless, this capability may be useful for breaking weak constructions that leak cross-session secrets to some recipients. As in eCK, the adversary can corrupt some of the private keys, as long as doing so is not enough to derive the target private key using BRAKEM.Decapsulate honestly (i.e., the target key is *fresh*). Unlike eCK, the BRAKEM security game implicitly encodes the notion of freshness in the challenge and derivation tables

[2] For example, this is the approach used in the definition of "soundness" for DREAD schemes, as discussed in Section 5.3.2.

and the behavior of the oracles. Importantly, it is possible to follow the implications in the tables in polynomial time when checking to see if the adversary has attacked a fresh key.[3] The game also does not need to differentiate between corruption of long-term keys and corruption of ephemeral keys because BRAKEM does not provide authentication, and thus requires no long-term identity keys. The other significant difference from eCK is that the key secrecy game is defined as a computational problem. This is a weaker guarantee than a decisional version of the game, but it is sufficient in practice if BRAKEM secret keys are only used as inputs to random oracles (or as targets of recursive BRAKEM transmissions).

8.1.1 Overview of the General Construction

Ultimately, all of the BRAKEM constructions use the same high-level technique. For each target ring, the private key is encrypted to each public key in the ring using ElGamal encryption. Two NIZKPKs are generated for each target ring in order to provide public verifiability. The first NIZKPK proves that all ElGamal ciphertexts for the ring encrypt the same plaintext. The second NIZKPK proves that the plaintext is the private key corresponding to the new public key. These NIZKPKs can be constructed with a variety of different zero-knowledge proof systems, each providing different performance and security characteristics. In practice, the NIZKPK proving that the correct private key was encrypted is far more difficult to construct. The remainder of this chapter presents several options.

8.2 BRAKEM from Double Discrete Logarithms

One method for constructing the NIZKPKs for the BRAKEM scheme is to operate within specially chosen groups for which it is possible to efficiently prove knowledge of a Double Discrete Logarithm (DDL). This section describes two constructions using this method, including a novel NIZKPK for proving the equality of multiple double discrete logarithms. This NIZKPK is complex and incorporates a number of new techniques of independent interest. For this reason, this section builds up the necessary techniques slowly by presenting several intermediate NIZKPKs.

[3] The derivation table encodes a directed acyclic graph with a polynomial number of vertices: the vertex count is polynomial in the number of calls to $O_{\text{Encapsulate}}$, which is only invoked by the PPT adversary. Checking to see if the statements in the corruption log imply Corrupt_{pk} = TRUE is equivalent to solving the reachability problem for the graph. This problem can be solved in polynomial time using the Floyd-Warshall algorithm [Flo62], among others.

8.2.1 Notation

Let p_0, p_1, p_2, and p_3 be large primes such that the group $\mathbb{G}_0 = \langle g_0 \rangle$ is a multiplicative subgroup in $\mathbb{Z}^*_{p_0}$ with order p_1, the group $\mathbb{G}_1 = \langle g_1 \rangle$ is a multiplicative subgroup in $\mathbb{Z}^*_{p_1}$ with order p_2, and the group $\mathbb{G}_2 = \langle g_2 \rangle$ is a multiplicative subgroup in $\mathbb{Z}^*_{p_2}$ with order p_3. The primes should be large enough that the Discrete Logarithm (DL) problem is difficult in $\mathbb{G}_0$ and $\mathbb{G}_1$, and the DDH problem is also difficult in $\mathbb{G}_2$. The primes should have the form $p_0 = 2p_1 + 1$, $p_1 = 2p_2 + 1$, and $p_2 = 2qp_3 + 1$, where q is a large cofactor such that $q \geq 2^{4\lceil \log_2(p_3) \rceil + 1}$. As an example at the 128-bit security level, p_3, p_2, p_1, and p_0 may be 256, 3072, 3073, and 3074 bits, respectively. The large cofactor $2q$ is a standard optimization to improve the efficiency of exponentiation in $\mathbb{G}_2$, but it is also necessary in order to use one of the highly efficient NIZKPKs in this construction.[4] Finding secure parameters for p_0, p_1, p_2, and p_3 that meet these constraints is efficient but non-trivial. Section 8.3.1.2 describes methods for generating appropriate primes. Let "$\mathbb{G}_i$" denote a unique encoding of the definition of group $\mathbb{G}_i$ for $i \in \{0, 1, 2\}$, including all relevant parameters.

8.2.2 Previously Known Zero-Knowledge Proofs

This section covers previously known NIZKPK constructions that are either useful in the BRAKEM constructions, or serve as an intermediate step to understanding the new NIZKPKs introduced later.

8.2.2.1 DL: Discrete Logarithm

The NIZKPKs used to construct BRAKEM in this setting are composed of simpler NIZKPKs that all derive from the basic Schnorr protocol [Sch91] for proving knowledge of a DL. Recall that the Schnorr protocol in $\mathbb{G}_2$ proves the following statement, specified in Camenisch-Stadler notation [CS97]:

$$PK\{(\alpha) : x = {g_2}^{\alpha}\}$$

In other words, the prover demonstrates to the verifier that it knows a witness α such that $\alpha = \text{dlog}_{g_2}(x)$ for some public $x \in \mathbb{G}_2$. While the Schnorr protocol itself is interactive, it can be transformed into the NIZKPK below in the random oracle model using the Fiat-Shamir transform [FS87]. The prover $\mathcal{P}$ proceeds as follows:

1. Choose $r \xleftarrow{\$} [2, p_3)$

[4] The DDL-based constructions require that q is large. Specifically, the approach does not work if $q = 1$, which is historically a common choice.

2. Compute $t' \leftarrow {g_2}^r$
3. Compute $c' \leftarrow H(\text{“}\mathbb{G}_2\text{”}\|x\|t')$ where H is a cryptographic hash function modeled by a random oracle
4. Compute $s \leftarrow r - c'\alpha \pmod{p_3}$
5. The proof is $\pi = (c', s)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $x \in \mathbb{G}_2$ and abort otherwise
2. Compute $t \leftarrow {g_2}^s \cdot x^{c'}$
3. Compute $c \leftarrow H(\text{“}\mathbb{G}_2\text{”}\|x\|t)$
4. Accept the proof if and only if $c = c'$

8.2.2.2 DLEQ: Discrete Logarithm Equality

A simple modification to the DL proof in Section 8.2.2.1 produces a proof that shows that two discrete logarithms in the same group[5] are equal. This variant, called a proof of Discrete Logarithm Equality (DLEQ) and denoted by $\text{DLEQ}\{\alpha : (h_1, x_1) \approx (h_2, x_2)\}$, proves the following statement:

$$\text{DLEQ}\{\alpha : (h_1, x_1) \approx (h_2, x_2)\} := PK\{(\alpha) : x_1 = {h_1}^\alpha \wedge x_2 = {h_2}^\alpha\}$$

In the proof statement, h_1, h_2, x_1, and x_2 are all publicly known elements of $\mathbb{G}_2$. The first construction for this proof was proposed by Chaum and Pedersen [CP93]. The prover $\mathcal{P}$ proceeds as follows:

1. Choose $r \xleftarrow{\$} [2, p_3)$
2. Compute ${t_1}' \leftarrow {h_1}^r$ and ${t_2}' \leftarrow {h_2}^r$
3. Compute $c' \leftarrow H(\text{“}\mathbb{G}_2\text{”}\|x_1\|x_2\|h_1\|h_2\|{t_1}'\|{t_2}')$ where H is a cryptographic hash function modeled by a random oracle

[5] The proof technique also works for two elements from different groups if the groups are known to have the same order. This chapter focuses on the case where both elements are in $\mathbb{G}_2$.

4. Compute $s \leftarrow r - c'\alpha \pmod{p_3}$
5. The proof is $\pi = (c', s)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $x_1, x_2, h_1, h_2 \in \mathbb{G}_2$ and abort otherwise
2. Compute $t_1 \leftarrow {h_1}^s \cdot {x_1}^{c'}$ and $t_2 \leftarrow {h_2}^s \cdot {x_2}^{c'}$
3. Compute $c \leftarrow H(\text{“}\mathbb{G}_2\text{”} \| x_1 \| x_2 \| h_1 \| h_2 \| t_1 \| t_2)$
4. Accept the proof if and only if $c = c'$

8.2.2.3 BDLEQ: Batch Discrete Logarithm Equality

The BRAKEM construction requires a proof that many elements share a common exponent with respect to different public bases. While it is possible to prove this for n discrete logarithms using $n-1$ *DLEQ* NIZKPKs, it is more efficient to use a “batch proof”. A Batch Discrete Logarithm Equality (BDLEQ) proof, denoted by $\text{BDLEQ}\{\alpha : (h_1, x_1) \approx \ldots \approx (h_n, x_n)\}$, is a batch proof of the following statement:

$$\text{BDLEQ}\{\alpha : (h_1, x_1) \approx \ldots \approx (h_n, x_n)\} := PK\{(\alpha) : \forall_{i \in [1,n]} x_i = h_i^\alpha\}$$

The group elements for this statement are all in $\mathbb{G}_2$. The most efficient proof technique for this statement was introduced by Peng et al. [PBD07, Fig. 4] and later refined using a modified random multiexponentiation test by Henry [Hen14, Fig. 3.4]. The protocol is implicitly parameterized by a security parameter λ_0. The protocol's soundness error is $2^{-\lambda_0}$. While this protocol is not a Σ-protocol, it is a public-coin honest-verifier zero-knowledge argument of knowledge, and thus the Fiat-Shamir transform applies. The prover $\mathcal{P}$ produces the NIZKPK as follows:

1. Compute $e = H_1(x_1 \| h_1 \| \ldots \| x_n \| h_n)$ where H_1 is a cryptographic hash function with $\lambda_0 \cdot (n-1)$ bits of output
2. Split e into $n-1$ parts such that for each $i \in [2, n]$, the part e_i is in $[0, 2^{\lambda_0})$
3. Choose $r \xleftarrow{\$} \mathbb{Z}_{p_3}$
4. Compute $t_1' \leftarrow h_1^r$

5. Compute $t_2' \leftarrow (\prod_{i=2}^{n} h_i^{e_i})^r$
6. Compute $c' \leftarrow H_2(x_1 \| h_1 \| \ldots \| x_n \| h_n \| t_1' \| t_2')$ where H_2 is a cryptographic hash function with λ_0 bits of output modeled by a random oracle
7. Compute $s \leftarrow r - c'\alpha \pmod{p_3}$
8. The proof is $\pi = (c', s)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $h_1, x_1, \ldots, h_n, x_n \in \mathbb{G}_2$ and abort otherwise
2. Compute $e = H_1(x_1 \| h_1 \| \ldots \| x_n \| h_n)$ and split e into e_i for $i \in [2, n]$
3. Compute $t_1 \leftarrow h_1^s \cdot x_1^{c'}$
4. Compute $t_2 \leftarrow (\prod_{i=2}^{n} h_i^{e_i})^s \cdot (\prod_{i=2}^{n} x_i^{e_i})^{c'}$
5. Compute $c \leftarrow H_2(x_1 \| h_1 \| \ldots \| x_n \| h_n \| t_1 \| t_2)$
6. Accept the proof if and only if $c = c'$

The asymptotic efficiently advantage of this approach over naïvely producing $n - 1$ *DLEQ* proofs is due to the ability to quickly compute t_2' and t_2. The e_i exponents are all small compared to p_3, and so the t_2' and t_2 products can be efficiently computed using a multiexponentiation algorithm.

This proof system is correct, sound, and zero-knowledge. However, Henry's soundness proof for this BDLEQ protocol [Hen14, Th. 3.17] contains an oversight. The theorem claims that the BDLEQ construction is component-wise 2-extractible, but unlike the proofs given for related schemes, there is no way to actually activate that scenario for the BDLEQ construction. Specifically, since the e_i values are determined *prior* to the prover's selection of r, there is no way for an extractor to rewind the e_i values while keeping the same t_1'. The proof of soundness should actually show that the BDLEQ construction is 2-extractible using the same proof technique as Peng et al. [PBD07, Th. 3]: if the verifier accepts, then the discrete logarithm of each component with respect to its associated base must be the same. This result can be combined with the normal knowledge extractor for the Schnorr proof system [Sch91] to show that a shared discrete logarithm can be recovered with a single rewind of c'.

8.2.2.4 DDL: Double Discrete Logarithm

The group definitions in Section 8.2.1 are carefully chosen so that the group operation for $\mathbb{G}_2$ can performed "in the exponent" of $\mathbb{G}_1$. Settings of this type have long been known to lend themselves to efficient NIZKPKs proving statements about the exponents of $\mathbb{G}_1$ elements. Specifically, it is possible to produce a NIZKPK of a DDL [Sta96]. DDL$\{\alpha : h_1, h_2, x\}$ denotes a proof of the following statement:

$$\text{DDL}\{\alpha : h_1, h_2, x\} := PK\{(\alpha, \beta) : x = {h_1}^{\beta} \wedge \beta = {h_2}^{\alpha}\}$$

In this equation, h_1 and x are public elements of $\mathbb{G}_1$ and h_2 is a public element of $\mathbb{G}_2$.

Several papers have examined constructions for DDL proofs and their applications. The first DDL proof technique was introduced by Stadler [Sta96]: a simple cut-and-choose protocol. The original application for this proof was publicly verifiable secret sharing (PVSS), which requires additional machinery. The original proof was used to produce a verifiable ElGamal encryption of an ElGamal secret key. Camenisch and Stadler [CS97] simplified the proof into one for the aforementioned DDL statement in order to construct a group signature scheme. Later, Schoenmakers [Sch99] proposed a method to avoid the overhead of the cut-and-choose protocol for PVSS, while also noting that the DDL setting prevents the use of more efficient elliptic curve groups.[6] Nakanishi et al. [NHS99] used Stadler's verifiable encryption in an RSA setting to implement digital cash. Camenisch and Shoup [CS03a] constructed verifiable encryption without the need for the cut-and-choose proof by relying on the strong RSA assumption. This scheme is far more efficient, but it is not practical for use in BRAKEM because any recursive scheme with RSA-based key pairs cannot implement BRAKEM without verifiably generating a secure RSA modulus; while NIZKPKs to accomplish this are known [CM99a], they are too expensive. This cost can be eliminated by using a single RSA group produced as part of a trusted setup, but any such group must be very large; the BRAKEM construction in this section is more efficient. Konoma et al. [KMS05] were the first to analyze the security of the DDL problem. Their results show that the DDL problem is at least as hard as the DL problem. Tate and Vishwanathan [TV09] noted that the cut-and-choose protocol can be eliminated using trusted hardware. D'Souza et al. [DJMP11] described an alternative PVSS scheme constructed using cryptographic pairings with a clever variant of ElGamal encryption. Unfortunately, this technique cannot be used to construct a BRAKEM scheme because it cannot be made recursive.

[6] Using an elliptic curve for the "exponent group" with generator h_2 does not work because mapping curve points to integers breaks the structure necessary for the proof system. While an elliptic curve may be used for the "outer group" with generator h_1, the size of the exponent group negates any performance benefits.

Consequently, the best previously known technique for constructing DDL proofs for use in BRAKEM remains the relatively slow cut-and-choose protocol first described in isolation by Camenisch and Stadler [CS97]. This NIZKPK was constructed using the Fiat-Shamir transform [FS87]. The prover $\mathcal{P}$ produces the NIZKPK as follows:

1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ for $1 \leq i \leq \ell$
2. Compute $t'_i \leftarrow h_1^{h_2^{r_i}}$ for $1 \leq i \leq \ell$
3. Compute $c' \leftarrow H(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|h_1\|h_2\|x\|t'_1\|\ldots\|t'_\ell)$, where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle
4. Compute for $1 \leq i \leq \ell$:

$$s_i \leftarrow \begin{cases} r_i & \text{if } c'[i] = 0 \\ r_i - \alpha \pmod{p_3} & \text{otherwise} \end{cases}$$

5. The proof is $\pi = (c', s_1, \ldots, s_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $h_1, x \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$ and abort otherwise
2. Compute for $1 \leq i \leq \ell$:

$$t_i \leftarrow \begin{cases} h_1^{h_2^{s_i}} & \text{if } c'[i] = 0 \\ x^{h_2^{s_i}} & \text{otherwise} \end{cases}$$

3. Compute $c \leftarrow H(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|h_1\|h_2\|x\|t_1\|\ldots\|t_\ell)$
4. Accept the proof if and only if $c = c'$

The prover can cheat with probability $2^{-\ell}$, so ℓ must be large enough for the security parameter.

Section 8.2.6 presents a novel technique to dramatically improve the performance of this cut-and-choose protocol in a manner that preserves its applicability to recursive key encapsulation.

8.2.2.5 DDLEQ: Double Discrete Logarithm Equality

As part of the BRAKEM construction, the DDL proof in Section 8.2.2.4 must be augmented to prove that the double discrete logarithm is equal to the (single) discrete logarithm for another element. A Double Discrete Logarithm Equality (DDLEQ) proof, denoted by DDLEQ$\{\alpha : (x, k) \approx y\}$, is a proof of the following statement:

$$\text{DDLEQ}\{\alpha : (x, k) \approx y\} := PK\{(\alpha, \beta) : k = g_1^{\beta} \wedge \beta = x^{\alpha} \wedge y = g_2^{\alpha}\}$$

In this proof statement, x and y are public elements in $\mathbb{G}_2$, and k is a public element in $\mathbb{G}_1$. While the construction can be easily derived from Stadler's original DDL proof [Sta96], the first time that this statement was mentioned was by Nakanishi et al. [NHS99]. Unfortunately, it does not appear to be possible to combine a standard Schnorr discrete log proof [Sch91] with the cut-and-choose protocol as in the DLEQ construction from Section 8.2.2.2. Chaum's original cut-and-choose protocol [CEG88] can be used instead. A similar technique was used by Stadler [Sta96] for verifiable ElGamal encryption, and Brecher et al. [BBM09, §4.1] later used a construction for this proof in the context of group key exchanges. The combined proof is produced by the prover $\mathcal{P}$ as follows:

1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ for $1 \leq i \leq \ell$
2. Compute $t'_{1,i} \leftarrow g_1^{x^{r_i}}$ and $t'_{2,i} \leftarrow g_2^{r_i}$ for $1 \leq i \leq \ell$
3. Compute $c' \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| x \| k \| y \| t'_{1,1} \| t'_{2,1} \| \ldots \| t'_{1,\ell} \| t'_{2,\ell})$, where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle
4. Compute for $1 \leq i \leq \ell$:

$$s_i \leftarrow \begin{cases} r_i & \text{if } c'[i] = 0 \\ r_i - \alpha \pmod{p_3} & \text{otherwise} \end{cases}$$

5. The proof is $\pi = (c', s_1, \ldots, s_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $k \in \mathbb{G}_1$ and $x, y \in \mathbb{G}_2$ and abort otherwise

2. Compute for $1 \leq i \leq \ell$:

$$t_{1,i} \leftarrow \begin{cases} g_1^{x^{s_i}} & \text{if } c'[i] = 0 \\ k^{x^{s_i}} & \text{otherwise} \end{cases} \qquad t_{2,i} \leftarrow \begin{cases} g_2^{s_i} & \text{if } c'[i] = 0 \\ y \cdot g_2^{s_i} & \text{otherwise} \end{cases}$$

3. Compute $c \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| x \| k \| y \| t_{1,1} \| t_{2,1} \| \ldots \| t_{1,\ell} \| t_{2,\ell})$

4. Accept the proof if and only if $c = c'$

8.2.3 CG-DLEQ: Cross-Group Discrete Logarithm Equality

This section discusses a new NIZKPK that is necessary to understand the DDL-based BRAKEM constructions. Using the DDLEQ NIZKPK described in Section 8.2.2.5, it is possible to verifiably encrypt the exponent for an element of $\mathbb{G}_1$ using the ElGamal cryptosystem. However, in order to create a recursive scheme, the proof must also show that the exponent of this $\mathbb{G}_1$ element is the same as the exponent of a $\mathbb{G}_2$ element, thereby “moving” the secret back into the correct group. This is called a proof of Cross-Group Discrete Logarithm Equality (CG-DLEQ). One might be tempted to simply use the DLEQ NIZKPK described in Section 8.2.2.2, proving the following statement with respect to the group setting defined in Section 8.2.1:

$$PK\{(\alpha) : x_1 = {g_1}^{\alpha} \wedge x_2 = {g_2}^{\alpha}\}$$

At first glance, using the DLEQ construction in Section 8.2.2.2 with $h_1 = g_1$ and $h_2 = g_2$ seems to prove that $\text{dlog}_{g_1}(x_1) = \text{dlog}_{g_2}(x_2)$, but this is not the case. Instead, it actually shows that $\alpha = \text{dlog}_{g_1}(x_1) \pmod{p_2}$ and $\alpha = \text{dlog}_{g_2}(x_2) \pmod{p_3}$. This statement is not actually meaningful, because if the prover knows the discrete logarithms of x_1 and x_2, then it is always possible to find a value of α that satisfies the equations using the Chinese remainder theorem, even if the discrete logarithms are not “equal”.

Consider the implications of the proof statement for a witness α within various ranges. If $0 \leq \alpha < p_3$, then the discrete logarithms are exactly “equal” in the sense that neither exponentiation “wraps around” its subgroup. If $p_3 \leq \alpha < p_2$, then the discrete logarithms are “equal” when the discrete logarithm of x_1 is taken modulo p_3 (i.e., the exponentiation for x_2 “wraps around”, but the exponentiation for x_1 does not). If $\alpha \geq p_2$, then there is no intuitive notion of “equality” for the discrete logarithms. Based on these observations, the NIZKPK can be made useful by adding a range restriction:

$$PK\{(\alpha) : x_1 = {g_1}^{\alpha} \wedge x_2 = {g_2}^{\alpha} \wedge 0 < \alpha < p_3\}$$

There are two standard techniques for adding this range restriction for α: use two DLEQ proofs to show that x_1 and x_2 both have the same discrete logarithm as an element in a group for which the prover does not know the order (e.g., modulo an RSA number generated by the verifier or the trusted setup), or use special-purpose "range proofs" to show that α is within the appropriate range [NBMV99; CM99b; Bou00]. Unfortunately, range proofs are too inefficient for secure group messaging applications. Using a group with the strong RSA assumption is an option, but it is possible to achieve better performance in the setting for the BRAKEM construction.

The new approach to implement CG-DLEQ described in this section is similar to the pioneering work of Chan et al. [CFT98, §4], where a similar technique was used to construct range-bounded commitments. The fact that $p_2 \gg p_3$ in the setting of interest (Section 8.2.1) means that it is acceptable to have a sound proof of a looser range than is used by honest provers. Let ℓ and ℓ' be security parameters, where ℓ is the security level of the discrete logarithm problem and ℓ' controls the proof's soundness. In typical settings, $\ell' \approx 3\ell$; the precise method to find the optimal value for ℓ' is discussed after the proof definition. CG-DLEQ denotes a proof of the following statement:

$$\text{CG-DLEQ}\{\alpha : x_1 \approx x_2\} := PK\{(\alpha) : x_1 = {g_1}^{\alpha} \wedge x_2 = {g_2}^{\alpha} \wedge -2^{\ell'+1} < \alpha < 2^{\ell'+1}\}$$

Since $p_2 > 2^{\ell'+2}$, proving this statement shows that the discrete logarithms are intuitively "equal" when taken modulo p_3. However, the efficient construction for this statement only provides statistical zero-knowledge for witnesses that are within the expected range $[0, p_3)$. This is acceptable for the intended application. To prove the statement, the prover $\mathcal{P}$ proceeds as follows:

1. Choose $R_i \xleftarrow{\$} \mathbb{Z}_{2^{\ell'}}$ for $1 \le i \le \ell$
2. Compute ${T_{1,i}}' \leftarrow {g_1}^{R_i}$ and ${T_{2,i}}' \leftarrow {g_2}^{R_i}$ for $1 \le i \le \ell$
3. Compute $c' \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| x_1 \| x_2 \| {T_{1,1}}' \| {T_{2,1}}' \| \ldots \| {T_{1,\ell}}' \| {T_{2,\ell}}')$ where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle
4. Compute for $1 \le i \le \ell$:

$$S_i \leftarrow \begin{cases} R_i & \text{if } c'[i] = 0 \\ R_i + \alpha \text{ (in } \mathbb{Z}) & \text{otherwise} \end{cases}$$

5. The proof is $\pi = (c', S_1, \ldots, S_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $x_1 \in \mathbb{G}_1$ and $x_2 \in \mathbb{G}_2$ and abort otherwise
2. Verify that $0 < S_i < 2^{\ell'+1}$ for $1 \leq i \leq \ell$ and abort otherwise
3. Compute for $1 \leq i \leq \ell$:

$$T_{1,i} \leftarrow \begin{cases} {g_1}^{S_i} & \text{if } c'[i] = 0 \\ {x_1}^{-1} \cdot {g_1}^{S_i} & \text{otherwise} \end{cases} \qquad T_{2,i} \leftarrow \begin{cases} {g_2}^{S_i} & \text{if } c'[i] = 0 \\ {x_2}^{-1} \cdot {g_2}^{S_i} & \text{otherwise} \end{cases}$$

4. Compute $c \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| x_1 \| x_2 \| T_{1,1} \| T_{2,1} \| \ldots \| T_{1,\ell} \| T_{2,\ell})$
5. Accept the proof if and only if $c = c'$

Note that when $\mathcal{P}$ computes a response S_i during step 4 when $c'[i] = 1$, the value of S_i is computed in $\mathbb{Z}$ and *not* modulo p_3. The range check by the verifier in step 2 provides the range guarantee in the proof statement: given two responses for the same commitment, a knowledge extractor can subtract them to recover a witness in the range. While the proof statement permits a wide range for α, honest provers must only use this proof when $\alpha \in \mathbb{Z}^*_{p_3}$ to avoid leaking information about α; the proof is sound for $\alpha \in (-2^{\ell'+1}, 2^{\ell'+1})$, but it is only zero-knowledge for $\alpha \in [0, p_3)$.

An important consideration to achieve the zero-knowledge property is the size of ℓ'. The NIZKPK can be shown to be zero-knowledge by constructing a simulator that chooses values for each S_i uniformly at random from $\mathbb{Z}_{2^{\ell'}}$, computes the expected $T'_{1,i}$ and $T'_{2,i}$ values, and then programs the random oracle. Camenisch [Cam98, Th. 5.1] showed that the distribution of $R_i + \alpha$ values created by this simulator are statistically indistinguishable from values produced by honest provers (i.e., provers using $\alpha \in \mathbb{Z}^*_{p_3}$) with probability $1 - 2^{-\ell}$ when $\ell' \geq \upsilon + 2\ell$, where υ is the minimum value satisfying $1 - (1 - 2/2^{\upsilon})^{\ell} \leq 2^{-\ell}$. For $\ell = 128$, this implies that $\ell' \geq 392$ must hold in order for the distributions to be statistically indistinguishable. Given that the proof demonstrates knowledge of α within $(-2^{\ell'+1}, 2^{\ell'+1})$ and $p_2 \geq 2^{\ell'+2}$, α cannot be within the range where the Chinese remainder theorem permits equivocation.[7]

This construction for CG-DLEQ is a cut-and-choose protocol. It is possible to implement it far more efficiently by adopting the technique used in the basic Schnorr protocol [Sch91], like the DL proof in Section 8.2.2.1, and adjusting ℓ' appropriately. This is similar to the approach used by Chan et al. [CFT98, §4] in their range-bounded commitments. However, this section presented the cut-and-choose variant because it is amenable to optimization when combined with the other required NIZKPKs later in this chapter.

[7] Notably, if the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ defined in Section 8.2.1 were instead chosen to use a safe prime p_2, then the proof would not be applicable.

8.2.4 BRAKEM Constructions

A general overview of the high-level construction for BRAKEM was given in Section 8.1.1. It is now possible to describe DDL-based constructions. The constructions are parameterized by a public key x called a "burner key". x is an element of $\mathbb{G}_2$ chosen using a cryptographic hash of identifying information for a group conversation, and therefore the corresponding private key is unknown. The presence of x improves the practical efficiency of the scheme, because it allows larger exponentiation tables to be precomputed. The downside is that computing the discrete logarithm of x with respect to g_2 compromises *all* conversations using the burner key. If this is a concern for a given application, the protocol can be adjusted to eliminate x at the cost of performance. The single point of failure can be mitigated somewhat by ensuring that each conversation has a distinct burner key, and possibly replacing the burner key over time.[8] Section 8.3.1.3 discusses security policies for the burner key in greater depth.

There are two DDL-based constructions discussed in this section. Section 8.2.4.1 introduces a construction using only standard security assumptions. Section 8.2.4.2 introduces a construction that is more than twice as efficient, but requires an unusual security assumption. An extremely optimized variant of the second construction is introduced later in Section 8.2.8.

8.2.4.1 A Conservative Construction: $\text{BRAKEM}_\star^{\text{2DDL}}$

The first DDL-based BRAKEM construction is denoted by $\text{BRAKEM}_x^{\text{2DDL}}$, where the parameter x is the burner key. In contexts where the chosen burner key does not matter, the scheme is written as $\text{BRAKEM}_\star^{\text{2DDL}}$. Recall that the goal of a BRAKEM scheme is to encapsulate new private keys to several sets of recipients; each of these sets is called a ring. For each ring, $\text{BRAKEM}_x^{\text{2DDL}}$ generates a random group element of $\mathbb{G}_2$ called u. It encrypts u^{-1} to the burner key x using ElGamal encryption in $\mathbb{G}_2$. It also encrypts u^{-1} to each recipient public key in the ring using the same ElGamal randomness. The ElGamal ciphertexts use u^{-1} as the plaintext instead of u itself because this is necessary in order for a subsequent NIZKPK to work. A BDLEQ (Section 8.2.2.3) proves that all of the ciphertexts encrypt the same plaintext u^{-1}. Next, it encrypts the inverse of the square of the new secret key $(sk_i^*)^{-2}$ to ${g_1}^u$ in $\mathbb{G}_1$ using ElGamal. It also attaches the values ${g_0}^{(sk_i^*)^2}$, ${g_2}^{(sk_i^*)^2}$, and ${g_2}^{sk_i^*} = pk_i^*$. A DDLEQ (Section 8.2.2.5) shows that the ciphertexts in $\mathbb{G}_2$ encrypt the inverse exponent for ${g_1}^u$, and a modified DDLEQ shows that the ciphertext in $\mathbb{G}_1$ encrypts the inverse exponent for ${g_0}^{(sk_i^*)^2}$. A modified CG-DLEQ (Section 8.2.3) shows that the exponent of the $\mathbb{G}_0$ element corresponds to the exponent in ${g_2}^{(sk_i^*)^2}$ taken modulo p_3 and that

[8] ^ Since the purpose of the burner key is to enable larger exponentiation tables to be precomputed, these tables need to be regenerated for each new burner key.

sk_i^* is within the expected range for $\mathbb{G}_2$. A DLEQ proof (Section 8.2.2.2) shows that the final two values in $\mathbb{G}_2$ are related by squaring. This process is repeated for each ring.

Two of the previously discussed NIZKPKs need to be modified to accommodate the presence of the $\mathbb{G}_0$ group. The modified DDLEQ proof, denoted by $\text{DDLEQ}_{\mathbb{G}_0,\mathbb{G}_1}$, is constructed identically to the DDLEQ proof in Section 8.2.2.5 except for using $\mathbb{G}_0$ where DDLEQ uses $\mathbb{G}_1$, and $\mathbb{G}_1$ where DDLEQ uses $\mathbb{G}_2$. Similarly, the modified CG-DLEQ proof, denoted by $\text{CG-DLEQ}_{\mathbb{G}_0}$, is constructed as in Section 8.2.3 except for taking its first input element from $\mathbb{G}_0$ instead of $\mathbb{G}_1$. The value of ℓ' must be increased for $\text{CG-DLEQ}_{\mathbb{G}_0}$ to accommodate the witness being from $\mathbb{Z}_{{p_3}^2}$ instead of $\mathbb{Z}_{p_3}$.

Once the NIZKPKs are all shown to be correct, sound, and zero-knowledge, the security of $\text{BRAKEM}_x^{\text{2DDL}}$ is intuitive. Since u is an element of $\mathbb{G}_2$, the ElGamal ciphertexts that transfer it to the recipients are IND-CPA (under the DDH assumption for $\mathbb{G}_2$) and thus leak no information about u. Since $p_1 = 2p_2 + 1$, x^2 is always a group element of $\mathbb{G}_1$ and thus the ElGamal ciphertext in $\mathbb{G}_1$ is also IND-CPA.[9] The zero-knowledge property of the NIZKPKs ensures that they do not leak any information about the secret exponents beyond the statements that they prove.

An additional optimization for this construction is replacing the ElGamal ephemeral key in $\mathbb{G}_1$ with ${g_1}^u$. This causes the ciphertext to become ${g_1}^{u^2} \cdot x^{-2}$, avoiding the need to transmit one $\mathbb{G}_1$ element. This optimization is secure if the decisional *squaring* Diffie-Hellman problem is hard—given $\mathbb{G}_1$ and ${g_1}^u$, distinguishing ${g_1}^{u^2}$ from a random element of $\mathbb{G}_1$. Maurer and Wolf [MW96, Th. 2] proved that this problem is hard if DDH is hard in a group with known order, such as $\mathbb{G}_1$.

To summarize, omitting the details of the NIZKPK that proves correctness of the ElGamal ciphertexts, $\text{BRAKEM}_x^{\text{2DDL}}$ implements the BRAKEM interface defined in Section 8.1 as follows:

- $\textbf{BRAKEM}_\star^{\textbf{2DDL}}\textbf{.KeyGen}(s) \rightarrow (pk, sk)$:

  > Choose sk in $\mathbb{Z}_{p_3}$ using randomness in s.
  > Compute $pk \leftarrow {g_2}^{sk}$.

- $\textbf{BRAKEM}_x^{\textbf{2DDL}}\textbf{.Encapsulate}(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$:

  > Derive all subsequently required randomness from s'.
  > **for each** $(1 \leq i \leq m)$ {
  > Choose ElGamal secret $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$.
  > Choose an element u_i of $\mathbb{G}_2$ uniformly at random.
  > Compute $y_i \leftarrow {g_2}^{r_i}$.

[9] Note that this construction requires the DDH problem to be difficult in $\mathbb{G}_1$, which is believed to hold for the given group definitions.

Compute ElGamal ciphertext $h_{1,i} \leftarrow x^{r_i} \cdot {u_i}^{-1}$.
for each $(1 \leq j \leq |R_i|)$ {
Compute ElGamal ciphertext $h_{1,i,j} \leftarrow {R_{i,j}}^{r_i} \cdot {u_i}^{-1}$.
}
Generate key pair $(pk_i^*, sk_i^*) \leftarrow \text{BRAKEM}_x^{\text{2DDL}}.\text{KeyGen}()$.
Compute $U_i \leftarrow {g_1}^{u_i}$.
Compute ElGamal ciphertext $h_{2,i} \leftarrow {g_1}^{{u_i}^2} \cdot {sk_i^*}^{-2}$.
Compute $k_{1,i} \leftarrow {g_0}^{sk_i^{*2}}$.
Compute $k_{2,i} \leftarrow {g_2}^{sk_i^{*2}}$.
Prove $\pi_{1,i} = \text{BDLEQ}\{r_i : (g_2, y_i) \approx ({R_{i,1}}/{x}, {h_{1,i,1}}/{h_{1,i}}) \approx \ldots \approx ({R_{i,|R_i|}}/{x}, {h_{1,i,|R_i|}}/{h_{1,i}})\}$.
Prove $\pi_{2,i} = \text{DDLEQ}\{r_i : (x, {U_i}^{h_{1,i}}) \approx y_i\}$.
Prove $\pi_{3,i} = \text{DDLEQ}_{\mathbb{G}_0,\mathbb{G}_1}\{u_i : (U_i, {k_{1,i}}^{h_{2,i}}) \approx U_i\}$.
Prove $\pi_{4,i} = \text{CG-DLEQ}_{\mathbb{G}_0}\{sk_i^{*2} : k_{1,i} \approx k_{2,i}\}$.
Prove $\pi_{5,i} = \text{DLEQ}\{sk_i^* : (pk_i^*, k_{2,i}) \approx (g_2, pk_i^*)\}$.
Assign $C_i = (y_i, h_{1,i,1}, \ldots, h_{1,i,|R_i|}, h_{2,i})$.
Assign $\pi_i = (h_{1,i}, U_i, k_{1,i}, k_{2,i}, \pi_{1,i}, \ldots, \pi_{5,i})$.
}
Assign $\pi = (\pi_1, \ldots, \pi_m)$.

- **$\text{BRAKEM}_x^{\text{2DDL}}$.Decapsulate**$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$:

if (**!** $((1 \leq i \leq m)\&\&(1 \leq j \leq |R_i|))$) **return** $\perp$.
Execute $v \leftarrow \text{BRAKEM}_x^{\text{2DDL}}.\text{Verify}(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$
$\hookrightarrow$ and record the interpretation of C_i.
if $(v = 0)$ **return** $\perp$.
Compute $u \leftarrow {{y_i}^{sk}}/{h_{1,i,j}}$.
Compute $k \leftarrow {{g_1}^{u^2}}/{h_{2,i}}$.
Compute k_1 and k_2, the two square roots of $k \pmod{p_3}$.[10]
if $(pk_i^* = {g_2}^{k_1})$ { **return** $sk_i^* = k_1$ } **else** { **return** $sk_i^* = k_2$ }.

- **$\text{BRAKEM}_x^{\text{2DDL}}$.Verify**$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow b$:

Interpret C_i for $1 \leq i \leq m$ and π as defined in $\text{BRAKEM}_x^{\text{2DDL}}$.Encapsulate.
if (interpretation of any C_i or π fails) **return** 0.

[10] ^ Due to the soundness of the underlying NIZKPKs, the successful return of $\text{BRAKEM}_x^{\text{2DDL}}$.Verify guarantees that k is a quadratic residue at this point in the algorithm.

for each $(1 \leq i \leq m)$ {
  if (any of $\pi_{1,i}, \ldots, \pi_{5,i}$ fail to verify) **return** 0.
}
return 1.

The definition of $\text{BRAKEM}_x^{\text{2DDL}}$.KeyGen is independent of the burner key. In other words, key pairs generated by the system can be reused in the context of different burner keys. However, decapsulation and verification must always be performed with respect to the same burner key used for encapsulation.

8.2.4.2 An Efficient Construction: $\text{BRAKEM}_\star^{\text{DDL}'}$

The BRAKEM construction in Section 8.2.4.1 relies on only conservative security assumptions. However, it is quite inefficient due to the presence of the $\text{DDLEQ}_{\mathbb{G}_0,\mathbb{G}_1}$ proof, which requires many expensive exponentiations in $\mathbb{G}_0$ and sending 128 large numbers from $\mathbb{Z}_{p_2}$. If it is possible to safely eliminate this proof, then the construction can be far more efficient.

$\text{BRAKEM}_\star^{\text{2DDL}}$ uses $\mathbb{G}_2$ as the "operating" group where keys are generated, with $\mathbb{G}_1$ and $\mathbb{G}_0$ being used purely for facilitating the NIZKPKs. Eliminating the expensive $\text{DDLEQ}_{\mathbb{G}_0,\mathbb{G}_1}$ proof requires avoiding the need for $\mathbb{G}_0$. This can be done if it is possible to securely encrypt the new private key (in $\mathbb{Z}_{p_3}$) using ElGamal in $\mathbb{G}_2$, instead of in $\mathbb{G}_1$. This is the core idea of the following efficient BRAKEM construction, denoted by $\text{BRAKEM}_\star^{\text{DDL}'}$:

- $\textbf{BRAKEM}_\star^{\text{DDL}'}\textbf{.KeyGen}(s) \rightarrow (pk, sk)$:

  Choose sk in $\mathbb{Z}_{p_3}$ using randomness in s.
  Compute $pk \leftarrow {g_2}^{sk}$.

- $\textbf{BRAKEM}_x^{\text{DDL}'}\textbf{.Encapsulate}(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$:

  Derive all subsequently required randomness from s'.
  for each $(1 \leq i \leq m)$ {
    Generate key pair $(pk_i^*, sk_i^*) \leftarrow \text{BRAKEM}_x^{\text{DDL}'}\text{.KeyGen}()$.
    Choose ElGamal secret $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$.
    Compute $y_i \leftarrow {g_2}^{r_i}$.
    Compute $\overline{sk_i^*} \leftarrow sk_i^{*-1} \pmod{p_2}$.
    Compute ElGamal ciphertext $h_i \leftarrow x^{r_i} \times \overline{sk_i^*} \pmod{p_2}$.

for each $(1 \leq j \leq |R_i|)$ {
  Compute ElGamal ciphertext $h_{i,j} \leftarrow {R_{i,j}}^{r_i} \times \overline{sk_i^*} \pmod{p_2}$.
}
Compute $k_i \leftarrow {g_1}^{sk_i^*}$.
Prove $\pi_{1,i} = \text{BDLEQ}\{r_i : (g_2, y_i) \approx ({}^{R_{i,1}}/_{x}, {}^{h_{i,1}}/_{h_i}) \approx \ldots \approx ({}^{R_{i,|R_i|}}/_{x}, {}^{h_{i,|R_i|}}/_{h_i})\}$.
Prove $\pi_{2,i} = \text{DDLEQ}\{r_i : (x, {k_i}^{h_i}) \approx y_i\}$.
Prove $\pi_{3,i} = \text{CG-DLEQ}\{sk_i^* : k_i \approx pk_i^*\}$.
Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
Assign $\pi_i = (h_i, k_i, \pi_{1,i}, \pi_{2,i}, \pi_{3,i})$.
}
Assign $\pi = (\pi_1, \ldots, \pi_m)$.

- **BRAKEM$_x^{\text{DDL}'}$.Decapsulate**$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$:

if (**!** $((1 \leq i \leq m)$**&&**$(1 \leq j \leq |R_i|))$) **return** $\perp$.
Execute $v \leftarrow \text{BRAKEM}_x^{\text{DDL}'}.\text{Verify}(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$
  $\hookrightarrow$ and record the interpretation of C_i.
if $(v = 0)$ **return** $\perp$.
Compute $k \leftarrow {}^{{y_i}^{sk}}/_{h_{i,j}}$.
return $sk_i^* = k \pmod{p_3}$.

- **BRAKEM$_x^{\text{DDL}'}$.Verify**$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow b$:

Interpret C_i for $1 \leq i \leq m$ and π as defined in $\text{BRAKEM}_x^{\text{DDL}'}$.Encapsulate.
if (interpretation of any C_i or π fails) **return** 0.
for each $(1 \leq i \leq m)$ {
  if (any of $\pi_{1,i}, \pi_{2,i}, \pi_{3,i}$ fail to verify) **return** 0.
}
return 1.

$\text{BRAKEM}_\star^{\text{DDL}'}$ is far more efficient than $\text{BRAKEM}_\star^{\text{2DDL}}$. However, there is one step that is not obviously secure: the publication of the ciphertext $h_i \leftarrow x^{r_i} \times \overline{sk_i^*} \pmod{p_2}$, where $\times$ denotes multiplication and $\overline{sk_i^*} = sk_i^{*-1} \pmod{p_2}$. While at first glance this appears to be a standard ElGamal ciphertext, ElGamal requires the plaintext space to be equal to the group (in this case, $\mathbb{G}_2$). In other words, ElGamal is only provably IND-CPA secure when encrypting group elements. Typically, real-world protocols work around this limitation by using ElGamal as part of a hybrid cryptosystem where the "plaintext" is a randomly chosen group element that is fed into a KDF to

produce the key for a DEM.[11] Another approach suggested by Chevallier-Mames et al. [CPP06] is to use a clever encoding of the DH shared secret in the ciphertext instead of using it directly. Unfortunately, the BRAKEM$^{\text{DDL}'}_{\star}$ construction requires the DH shared secret to be used "as-is" in order to make DDLEQ applicable. Similarly, sk_i^* must be used without any encoding in order for the CG-DLEQ proof to show that it corresponds to pk_i^*. These limitations present a problem for BRAKEM$^{\text{DDL}'}_{\star}$: because q is large, $\overline{sk_i^*}$ will only be a group element with negligible probability. ElGamal is provably not IND-CPA when the plaintext is not from the group: the adversary can win the IND-CPA game by simply dividing the ciphertext by the potential plaintexts and checking to see if the result is in $\mathbb{G}_2$. Consequently, proving the security of BRAKEM$^{\text{DDL}'}_{\star}$ requires additional hardness assumptions. These assumptions and the security proofs are sketched in Section 8.4.

Before discussing the security of BRAKEM$^{\text{DDL}'}_{\star}$, there are additional optimizations that can be applied to the protocol. These optimizations should be applied before proving the security of the scheme because they impact the details of the proof technique. Significant performance can be gained by batching all of the $\pi_{2,i}$ and $\pi_{3,i}$ NIZKPKs into a single large proof. The resulting proof is very complex and highly application-specific, but the techniques used to develop it are more generally applicable. The following sections incrementally introduce the combined NIZKPK, referred to simply as Π_0. The modified BRAKEM construction using the optimized proof is introduced in Section 8.2.8.

8.2.5 CG-DDLEQ: Cross-Group Double Discrete Logarithm Equality

In the construction for BRAKEM$^{\text{DDL}'}_{\star}$ in Section 8.2.4.2, the DDLEQ proof in $\pi_{2,i}$ combined with the CG-DLEQ proof in $\pi_{3,i}$ demonstrate that the ElGamal ciphertext h_i encrypts $\text{dlog}_{g_2}(pk_i^*)$. To begin the optimization process, these two proofs can be easily combined into a single proof that uses the same Fiat-Shamir [FS87] challenge. The proofs combine readily since they are both cut-and-choose constructions. While this standard optimization only saves the transmission of a single challenge value, it is an important first step for the batching process in subsequent optimizations. The combined proof is called a Cross-Group Double Discrete Logarithm Equality (CG-DDLEQ) and it establishes the following statement:

$$PK\{(\alpha,\beta,\gamma) : k^h = g_1^\beta \wedge \beta = x^\alpha \wedge y = g_2^\alpha \wedge k = {g_1}^\gamma \wedge z = {g_2}^\gamma \wedge -2^{\ell'+1} < \gamma < 2^{\ell'+1}\}$$

The security parameters ℓ and ℓ' are the same as in the CG-DLEQ construction (Section 8.2.3). The statement differs from the DDLEQ statement in Section 8.2.2.5 in that it explicitly includes

[11] This is the technique that was used in the DREAD construction in Section 5.3.2.

the public elements $k \in \mathbb{G}_1$ and $h \in \mathbb{Z}_{p_2}$ corresponding to k_i and h_i as used in $\text{BRAKEM}_\star^{\text{DDL}'}$. To prove the statement, the prover $\mathcal{P}$ proceeds as follows:

1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ for $1 \leq i \leq \ell$
2. Choose $R_i \xleftarrow{\$} \mathbb{Z}_{2^{\ell'}}$ for $1 \leq i \leq \ell$
3. Compute $t'_{1,i} \leftarrow g_1^{x^{r_i}}$ and $t'_{2,i} \leftarrow g_2^{r_i}$ for $1 \leq i \leq \ell$
4. Compute $T'_{1,i} \leftarrow g_1^{R_i}$ and $T'_{2,i} \leftarrow g_2^{R_i}$ for $1 \leq i \leq \ell$
5. Compute $c' \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| k \| h \| x \| y \| z \| t'_{1,1} \| t'_{2,1} \| T'_{1,1} \| T'_{2,1} \| \ldots \| t'_{1,\ell} \| t'_{2,\ell} \| T'_{1,\ell} \| T'_{2,\ell})$, where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle
6. Compute for $1 \leq i \leq \ell$:

$$s_i \leftarrow \begin{cases} r_i & \text{if } c'[i] = 0 \\ r_i - \alpha \pmod{p_3} & \text{otherwise} \end{cases} \qquad S_i \leftarrow \begin{cases} R_i & \text{if } c'[i] = 0 \\ R_i + \gamma \text{ (in } \mathbb{Z}) & \text{otherwise} \end{cases}$$

7. The proof is $\pi = (c', s_1, S_1, \ldots, s_\ell, S_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $k \in \mathbb{G}_1$ and $h, x, y, z \in \mathbb{G}_2$ and abort otherwise
2. Compute for $1 \leq i \leq \ell$:

$$t_{1,i} \leftarrow \begin{cases} g_1^{x^{s_i}} & \text{if } c'[i] = 0 \\ k^{x^{s_i}} & \text{otherwise} \end{cases} \qquad t_{2,i} \leftarrow \begin{cases} g_2^{s_i} & \text{if } c'[i] = 0 \\ y \cdot g_2^{s_i} & \text{otherwise} \end{cases}$$

$$T_{1,i} \leftarrow \begin{cases} g_1^{S_i} & \text{if } c'[i] = 0 \\ k^{-1} \cdot g_1^{S_i} & \text{otherwise} \end{cases} \qquad T_{2,i} \leftarrow \begin{cases} g_2^{S_i} & \text{if } c'[i] = 0 \\ z^{-1} \cdot g_2^{S_i} & \text{otherwise} \end{cases}$$

3. Verify that $0 \leq S_i < 2^{\ell'+1}$ for all $1 \leq i \leq \ell$ and abort otherwise
4. Compute $c \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| k \| h \| x \| y \| z \| t_{1,1} \| t_{2,1} \| T_{1,1} \| T_{2,1} \| \ldots \| t_{1,\ell} \| t_{2,\ell} \| T_{1,\ell} \| T_{2,\ell})$
5. Accept the proof if and only if $c = c'$

8.2.6 SDDLEQ: Simultaneous Double Discrete Logarithm Equality

The previously described DDL, DDLEQ, CG-DLEQ, and CG-DDLEQ schemes all suffer from the same performance flaw: they all require the verifier to perform $O(\ell)$ exponentiations in $\mathbb{G}_1$. These exponentiations are so computationally expensive that they dominate all other CPU time costs. This section presents a novel proof technique that dramatically improves the performance of DDLEQ proofs by eliminating most of the required $\mathbb{G}_1$ exponentiations. The technique shares similarities with both the random multiexponentiation technique used by Henry [Hen14, Fig. 3.4] in BDLEQ, and the Unruh transform [Unr15]. Interestingly, although the Unruh transform was developed to produce NIZKPKs with post-quantum security, a similar approach significantly improves the performance of DDL proofs in the classical random oracle model.

The modified NIZKPK is an alternate construction for the DDLEQ statement in Section 8.2.2.5, repeated here for convenience:

$$\text{DDLEQ}\{\alpha : (x, k) \approx y\} := PK\{(\alpha, \beta) : k = g_1^{\beta} \wedge \beta = x^{\alpha} \wedge y = g_2^{\alpha}\}$$

This new construction is called a *simultaneous proof of double discrete logarithm equality* (SDDLEQ). At a high level, the proof operates as follows:

1. The prover commits to all possible responses (for all possible challenge bits).
2. The verifier selects a random multiexponentiation.
3. The prover computes the requested multiexponentiation of committed secrets.
4. The verifier sends the challenge.
5. The prover reveals the selected responses.

This structure remains amenable to the Fiat-Shamir transform, but reduces the number of $\mathbb{G}_1$ exponentiations that the verifier needs to perform to $O(1)$; this results in a dramatic performance improvement in practice.

The prover $\mathcal{P}$ constructs the proof as follows:

1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ for $1 \leq i \leq \ell$
2. For $1 \leq i \leq \ell$, set $s'_{i,0} = r_i$ and $s'_{i,1} = r_i - \alpha \pmod{p_3}$

3. For $1 \leq i \leq \ell$, set $u'_i = H_0(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|i\|x\|k\|y\|s'_{i,0})$ and $v'_i = H_1(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|i\|x\|k\|y\|s'_{i,1})$, where H_0 and H_1 are independent cryptographic hash functions each having at least 2ℓ bits of output

4. Compute $e' = H_e(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|x\|k\|y\|u'_1\|v'_1\|\ldots\|u'_\ell\|v'_\ell)$ where H_e is an independent cryptographic hash function with at least $2\ell^2$ bits of output

5. Split e' into ℓ values in $[2, p_3)$: $e'_1, \ldots, e'_\ell$

6. Compute $t'_1 \leftarrow g_1^{x^{e'_1+r_1}+\cdots+x^{e'_\ell+r_\ell}}$

7. Compute $t'_2 \leftarrow g_2^{e'_1\cdot r_1+\cdots+e'_\ell\cdot r_\ell}$

8. Compute $c' \leftarrow H(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|x\|k\|y\|u'_1\|v'_1\|\ldots\|u'_\ell\|v'_\ell\|t'_1\|t'_2)$, where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle

9. For each $1 \leq i \leq \ell$, set $s_i \leftarrow s_{i,c'[i]}$

10. For each $1 \leq i \leq \ell$, if $c'[i] = 0$ then set $w_i \leftarrow v'_i$; otherwise, set $w_i \leftarrow u'_i$

11. The proof is $\pi = (c', s_1, w_1, \ldots, s_\ell, w_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $k \in \mathbb{G}_1$ and $x, y \in \mathbb{G}_2$ and abort otherwise

2. For each $1 \leq i \leq \ell$:

   a) If $c'[i] = 0$ then set $u_i \leftarrow H_0(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|i\|x\|k\|y\|s_i)$ and $v_i \leftarrow w_i$

   b) If $c'[i] = 1$ then set $u_i \leftarrow w_i$ and $v_i \leftarrow H_1(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|i\|x\|k\|y\|s_i)$

3. Compute $e = H_e(\text{“}\mathbb{G}_1\text{”}\|\text{“}\mathbb{G}_2\text{”}\|x\|k\|y\|u_1\|v_1\|\ldots\|u_\ell\|v_\ell)$

4. Split e into ℓ values in $[2, p_3)$: $e_1, \ldots, e_\ell$

5. Set $t_{1,0}$, $t_{1,1}$, $t_{2,0}$, and E to 0

6. For each $1 \leq i \leq \ell$:

   a) If $c'[i] = 0$ then set $t_{1,0} \leftarrow t_{1,0} + x^{e_i+s_i} \pmod{p_2}$

   b) If $c'[i] = 1$ then set $t_{1,1} \leftarrow t_{1,1} + x^{e_i+s_i} \pmod{p_2}$

c) In any case, set $t_{2,0} \leftarrow t_{2,0} + e_i \cdot s_i \pmod{p_3}$

d) If $c'[i] = 1$ then set $E \leftarrow E + e_i \pmod{p_3}$

7. Compute $t_1 \leftarrow {g_1}^{t_{1,0}} \cdot k^{t_{1,1}}$
8. Compute $t_2 \leftarrow {g_2}^{t_{2,0}} \cdot y^E$
9. Compute $c \leftarrow H(\text{“}\mathbb{G}_1\text{”} \| \text{“}\mathbb{G}_2\text{”} \| x \| k \| y \| u_1 \| v_1 \| \ldots \| u_\ell \| v_\ell \| t_1 \| t_2)$
10. Accept the proof if and only if $c = c'$

The correctness of this scheme is tedious but straightforward to prove. Note that in the t_2 case, the value of E accumulates all e_i values when $c'[i] = 1$. These are the cases in which $t_{2,0}$ will contain an extra $e_i \cdot \alpha$ term. Since the exponent of y is α, multiplying by y^E will cancel out these extra terms.

The soundness of the proof is non-trivial to prove. The rewinding knowledge extractor rewinds the prover and programs the random oracle's output c'; H_0 and H_1 are simply cryptographic hashes that are non-programmable. Given two responses for the same commitments where the challenges differ in at least one bit position, it can be shown that the extractor can always recover α by subtracting the different responses for this bit position. The main intuition required to prove this statement is that a malicious prover cannot choose any secrets based on the value of e. Because the verifier derives e from a cryptographic hash of the u_i and v_i values, which in turn derive from hashes of the s_i values and the proof statement, the prover has no ability to react to the selection of the e_i values. Using this fact, it can ultimately be shown that $e_1 \cdot (s_{1,0} - s_{1,1} - \text{dlog}_{g_2}(y)) + \cdots + e_\ell \cdot (s_{\ell,0} - s_{\ell,1} - \text{dlog}_{g_2}(y)) = 0$ implies that $s_{i,0} - s_{i,1} = \text{dlog}_{g_2}(y)$ for every $1 \leq i \leq \ell$. A similar statement can be made about the double discrete logarithm of k.

8.2.7 Combined NIZKPK: Π_0

Given the definitions of CG-DDLEQ and SDDLEQ in Section 8.2.5 and Section 8.2.6, it is now possible to discuss the actual NIZKPK used in the optimized BRAKEM construction. The NIZKPK, called Π_0, replaces every $\pi_{2,i}$ and $\pi_{3,i}$ for all $1 \leq i \leq m$ in $\text{BRAKEM}_x^{\text{DDL}'}$ (Section 8.2.4.2) with a single proof. Merging all of these statements into a single NIZKPK provides multiple dimensions that are amenable to batching optimizations.

Π_0 is a NIZKPK for the following statement:

$$\Pi_0\{(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m,\gamma_1,\ldots,\gamma_m) : x, (k_1,h_1,y_1,z_1) \approx \cdots \approx (k_m,h_m,y_m,z_m)\} :=$$
$$PK\{(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m,\gamma_1,\ldots,\gamma_m) :$$
$${k_1}^{h_1} = {g_1}^{\beta_1} \wedge \beta_1 = x^{\alpha_1} \wedge y_1 = {g_2}^{\alpha_1} \wedge k_1 = {g_1}^{\gamma_1} \wedge z_1 = {g_2}^{\gamma_1} \wedge -2^{\ell'+1} < \gamma_1 < 2^{\ell'+1} \wedge$$
$$\ldots$$
$${k_m}^{h_m} = {g_1}^{\beta_m} \wedge \beta_m = x^{\alpha_m} \wedge y_m = {g_2}^{\alpha_m} \wedge k_m = {g_1}^{\gamma_m} \wedge z_m = {g_2}^{\gamma_m} \wedge -2^{\ell'+1} < \gamma_m < 2^{\ell'+1}\}$$

This proof statement is a combination of the CG-DDLEQ proof from Section 8.2.5 and the SDDLEQ proof from Section 8.2.6 that has been batched across m statements simultaneously. To simplify, this proof demonstrates that for $1 \leq i \leq m$, h_i is an ElGamal ciphertext for the recipient public key x (the burner key) that encrypts the private key in z_i. The implementation of this proof is a combination of CG-DDLEQ and SDDLEQ with one twist: the random coefficients in the desired multiexponentiation are composed by multiplying smaller coefficients generated from a hash of the commitment. This allows related terms to conveniently simplify in the security proofs.

For a given statement, let "$\mathbb{G}$" = "$\mathbb{G}_1$"||"$\mathbb{G}_2$"$||\ell||\ell'||m||x||k_1||h_1||y_1||z_1||\ldots||k_m||h_m||y_m||z_m$. The prover $\mathcal{P}$ constructs the proof as follows:

1. Choose $r_{i,j} \xleftarrow{\$} \mathbb{Z}_{p_3}$ for $1 \leq i \leq \ell$ and $1 \leq j \leq m$
2. Choose $R_{i,j} \xleftarrow{\$} \mathbb{Z}_{2^{\ell'}}$ for $1 \leq i \leq \ell$ and $1 \leq j \leq m$
3. For $1 \leq i \leq \ell$ and $1 \leq j \leq m$:
   a) Set $s'_{i,j,0} \leftarrow r_{i,j}$
   b) Set $S'_{i,j,0} \leftarrow R_{i,j}$
   c) Set $s'_{i,j,1} \leftarrow r_{i,j} - \alpha_j \pmod{p_3}$
   d) Set $S'_{i,j,1} \leftarrow R_{i,j} + \gamma_j$ (in $\mathbb{Z}$)
4. For $1 \leq i \leq \ell$, set $u'_i = H_0(\text{"}\mathbb{G}\text{"}||i||s'_{i,1,0}||\ldots||s'_{i,m,0}||S'_{i,1,0}||\ldots||S'_{i,m,0})$ where H_0 is an independent cryptographic hash function with at least 2ℓ bits of output
5. For $1 \leq i \leq \ell$, set $v'_i = H_1(\text{"}\mathbb{G}\text{"}||i||s'_{i,1,1}||\ldots||s'_{i,m,1}||S'_{i,1,1}||\ldots||S'_{i,m,1})$ where H_1 is an independent cryptographic hash function with at least 2ℓ bits of output

6. Compute $H_e(\text{“}\mathbb{G}\text{”}\|u'_1\|v'_1\|\ldots\|u'_\ell\|v'_\ell)$ where H_e is an independent cryptographic hash function and extract from the hash output $\ell + 2m$ values in $[2, p_3)$: $e'_1, \ldots, e'_\ell, f'_1, \ldots, f'_m, F'_1, \ldots, F'_m$

7. Compute:

$$t' = g_1 \left\{ \begin{array}{c} x^{e'_1+f'_1+r_{1,1}} + \ldots + x^{e'_\ell+f'_1+r_{\ell,1}} + \\ \vdots \\ x^{e'_1+f'_m+r_{1,m}} + \ldots + x^{e'_\ell+f'_m+r_{\ell,m}} + \\ F'_1 \cdot (e'_1 \cdot R_{1,1} + \ldots + e'_\ell \cdot R_{\ell,1}) + \\ \vdots \\ F'_m \cdot (e'_1 \cdot R_{1,m} + \ldots + e'_\ell \cdot R_{\ell,m}) \end{array} \right\} \qquad T' = g_2 \left\{ \begin{array}{c} f'_1 \cdot (e'_1 \cdot r_{1,1} + \ldots + e'_\ell \cdot r_{\ell,1}) + \\ \vdots \\ f'_m \cdot (e'_1 \cdot r_{1,m} + \ldots + e'_\ell \cdot r_{\ell,m}) + \\ F'_1 \cdot (e'_1 \cdot R_{1,1} + \ldots + e'_\ell \cdot R_{\ell,1}) + \\ \vdots \\ F'_m \cdot (e'_1 \cdot R_{1,m} + \ldots + e'_\ell \cdot R_{\ell,m}) \end{array} \right\}$$

8. Compute $c' \leftarrow H(\text{“}\mathbb{G}\text{”}\|u'_1\|v'_1\|e'_1\|\ldots\|u'_\ell\|v'_\ell\|e'_\ell\|f'_1\|F'_1\|\ldots\|f'_m\|F'_m\|t'\|T')$, where H is a cryptographic hash function with at least ℓ bits of output modeled by a random oracle

9. For each $1 \le i \le \ell$:

   a) If $c'[i] = 0$ then set $w_i \leftarrow v'_i$ and for $1 \le j \le m$, set $s_{i,j} \leftarrow s'_{i,j,0}$ and $S_{i,j} \leftarrow S'_{i,j,0}$

   b) If $c'[i] = 1$ then set $w_i \leftarrow u'_i$ and for $1 \le j \le m$, set $s_{i,j} \leftarrow s'_{i,j,1}$ and $S_{i,j} \leftarrow S'_{i,j,1}$

10. The proof is $\pi = (c', s_{1,1}, \ldots, s_{\ell,m}, S_{1,1}, \ldots, S_{\ell,m}, w_1, \ldots, w_\ell)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $x \in \mathbb{G}_2$ and for each $1 \le j \le m$: $k_j \in \mathbb{G}_1$ and $y_j, z_j \in \mathbb{G}_2$ and abort otherwise

2. For each $1 \le i \le \ell$:

   a) If $c'[i] = 0$ then set $u_i \leftarrow H_0(\text{“}\mathbb{G}\text{”}\|i\|s_{i,1}\|\ldots\|s_{i,m}\|S_{i,1}\|\ldots\|S_{i,m})$ and $v_i \leftarrow w_i$

   b) If $c'[i] = 1$ then set $u_i \leftarrow w_i$ and $v_i \leftarrow H_1(\text{“}\mathbb{G}\text{”}\|i\|s_{i,1}\|\ldots\|s_{i,m}\|S_{i,1}\|\ldots\|S_{i,m})$

3. Compute $H_e(\text{“}\mathbb{G}\text{”}\|u_1\|v_1\|\ldots\|u_\ell\|v_\ell)$ and extract from the hash output $\ell+2m$ values in $[2, p_3)$: $e_1, \ldots, e_\ell, f_1, \ldots, f_m, F_1, \ldots, F_m$

4. Set t_0, T_0, E_1, E_2 and t_j for $1 \le j \le m$ to 0

5. For each $1 \le i \le \ell$:

   a) For each $1 \le j \le m$: set $t_0 \leftarrow t_0 + e_i \cdot F_j \cdot S_{i,j} \pmod{p_2}$

b) For each $1 \leq j \leq m$: set $T_0 \leftarrow T_0 + e_i \cdot (f_j \cdot s_{i,j} + F_j \cdot S_{i,j}) \pmod{p_3}$

c) If $c'[i] = 0$ then:

i. For $1 \leq j \leq m$: set $t_0 \leftarrow t_0 + x^{e_i + f_j + s_{i,j}} \pmod{p_2}$

d) If $c'[i] = 1$ then:

i. For $1 \leq j \leq m$: set $t_j \leftarrow t_j + x^{e_i + f_j + s_{i,j}} \pmod{p_2}$

ii. Set $E_1 \leftarrow E_1 - e_i \pmod{p_2}$

iii. Set $E_2 \leftarrow E_2 + e_i \pmod{p_3}$

e) Verify that $0 \leq S_{i,j} < 2^{\ell'+1}$ for all $1 \leq j \leq m$ and abort otherwise

6. Compute $t \leftarrow {g_1}^{t_0} \cdot \prod_{j=1}^{m} {k_j}^{h_j \cdot t_j + E_1 \cdot F_j}$

7. Compute $T \leftarrow {g_2}^{T_0} \cdot \prod_{j=1}^{m} ({y_j}^{E_2 \cdot f_j} \cdot {z_j}^{-E_2 \cdot F_j})$

8. Compute $c \leftarrow H(\text{“}\mathbb{G}\text{”} \| u_1 \| v_1 \| e_1 \| \ldots \| u_\ell \| v_\ell \| e_\ell \| f_1 \| F_1 \| \ldots \| f_m \| F_m \| t \| T)$

9. Accept the proof if and only if $c = c'$

Section 8.4 provides proofs of correctness, soundness, and zero knowledge for this NIZKPK.

8.2.8 An Optimized BRAKEM Construction: BRAKEM$^{\text{DDL}}_{\star}$

The highly specific NIZKPK introduced in Section 8.2.7 can replace the DDLEQ and CG-DLEQ proofs in BRAKEM$^{\text{DDL}'}_{\star}$ as defined in Section 8.2.4.2. The resulting BRAKEM construction is called BRAKEM$^{\text{DDL}}_{\star}$. It operates as follows:

- **BRAKEM$^{\text{DDL}}_{\star}$.KeyGen**$(s) \rightarrow (pk, sk)$:

  Choose sk in $\mathbb{Z}_{p_3}$ using randomness in s.
  Compute $pk \leftarrow {g_2}^{sk}$.

- **BRAKEM$^{\text{DDL}}_{x}$.Encapsulate**$(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$:

  Derive all subsequently required randomness from s'.
  for each $(1 \leq i \leq m)$ {
  Generate key pair $(pk_i^*, sk_i^*) \leftarrow$ BRAKEM$^{\text{DDL}}_{x}$.KeyGen().

Choose ElGamal secret $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$.
Compute $y_i \leftarrow {g_2}^{r_i}$.
Compute $\overline{sk_i^*} \leftarrow sk_i^{*-1} \pmod{p_2}$.
Compute ElGamal ciphertext $h_i \leftarrow x^{r_i} \times \overline{sk_i^*} \pmod{p_2}$.
for each $(1 \leq j \leq |R_i|)$ {
    Compute ElGamal ciphertext $h_{i,j} \leftarrow {R_{i,j}}^{r_i} \times \overline{sk_i^*} \pmod{p_2}$.
}
Compute $k_i \leftarrow {g_1}^{sk_i^*}$.
Prove $\pi_i = \text{BDLEQ}\{r_i : (g_2, y_i) \approx ({}^{R_{i,1}}/_{x}, {}^{h_{i,1}}/_{h_i}) \approx \ldots \approx ({}^{R_{i,|R_i|}}/_{x}, {}^{h_{i,|R_i|}}/_{h_i})\}$.
Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Prove $\pi_0 = \Pi_0\{(r_1, \ldots, r_m, x^{r_1}, \ldots, x^{r_m}, sk_1^*, \ldots, sk_m^*) :$
$x, (k_1, h_1, y_1, pk_1^*) \approx \cdots \approx (k_m, h_m, y_m, pk_m^*)\}$.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.

- **BRAKEM$_x^{\text{DDL}}$.Decapsulate**$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$:

if (**!** $((1 \leq i \leq m)$**&&**$(1 \leq j \leq |R_i|))$) **return** $\perp$.
Execute $v \leftarrow \text{BRAKEM}_x^{\text{DDL}}.\text{Verify}(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$
    $\hookrightarrow$ and record the interpretation of C_i.
if $(v = 0)$ **return** $\perp$.
Compute $k \leftarrow {}^{{y_i}^{sk}}/_{h_{i,j}}$.
return $sk_i^* = k \pmod{p_3}$.

- **BRAKEM$_x^{\text{DDL}}$.Verify**$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow b$:

Interpret C_i for $1 \leq i \leq m$ and π as defined in BRAKEM$_x^{\text{DDL}}$.Encapsulate.
if (interpretation of any C_i or π fails) **return** 0.
if (π_0 fails to verify) **return** 0.
for each $(1 \leq i \leq m)$ {
    if (π_i fails to verify) **return** 0.
}
return 1.

8.2.9 On the Need for CG-DLEQ

GKEs built with DH exchanges in tree arrangements are typically based on the pioneering work of Kim et al. [KPT00; KPT04]. This original scheme and derivations such as ART [CCG+18] (as described in Section 4.1) use a clever trick to avoid the need for two different groups: they use a group with prime order q for which group elements can be mapped to integers in $[1, q]$. Kim et al. [KPT04, §A.1] define such a group, $\mathbb{G}$, with generator $g = 2$ and prime order q such that $p = 2q + 1$ is a (safe) prime. The group operation $\cdot$ is defined as $a \cdot b = f(a \times b \pmod p)$, where $\times$ denotes multiplication and $f(x)$ is defined as follows:

$$f(x) = \begin{cases} x & \text{if } x \leq q \\ p - x & \text{if } q < x < p \end{cases}$$

The function f is a bijection when $p \equiv 3 \pmod 4$, which is the case when $p = 2q + 1$. The inclusion of f in the group operation ensures that each element of $\mathbb{G}$ can be written in $[1, q]$. This group can also be written as $\mathbb{G} = \mathbb{Z}_p^*/\langle -1 \rangle$, which contains elements of the form $\{-i, i\}$ for $i \in [1, q]$ and uses multiplication modulo p, ignoring the sign, as the group operation. One can use $\mathbb{G}$ to easily compute values like $g^{g^{xy}}$, which enables efficient "recursive" DH exchanges for use in tree-based GKEs. Groups of this type were first described by Chaum [Cha90].

Unfortunately, while these "Chaum groups" are useful for constructing GKEs, the Schnorr proof system [Sch91] and derivatives cannot prove statements about them. Brecher et al. [BBM09] erroneously attempted to use Chaum groups to add insider security to a GKE.[12] They use a DDLEQ proof, nearly the same as the one described in Section 8.2.2.5, to prove that a DH exchange was correctly used to derive a shared public key. Their proof statement [BBM09, §4.1] is as follows: given Chaum group $\mathbb{G}$ with generator g and public group elements y, $\widetilde{y_1}$, and $\widetilde{y_2}$, prove $PK\{(x) : \widetilde{y_1} = g^x \wedge \widetilde{y_2} = g^{y^x}\}$. This proof fails in a Chaum group because the function f does not have the required ring structure. Although exponentiation can be implemented through repeated application of the custom group operator, the effect on the value "in the exponent" will still be multiplication modulo q. Since this is not the group operation, the prover and verifier will end up computing different values.

The mismatch in the scheme defined by Brecher et al. occurs in cases where the i^{th} challenge bit is 1: the prover computes $t_{2,i} = g^{y^{\alpha_i}}$, but the verifier computes $\bar{t}_{2,i} = \widetilde{y_2}^{y^{s_i}}$. In a Chaum group,

[12] While Brecher et al. never fully describe the group used in their setting, they specify a group generated by a quadratic residue modulo a safe prime for which there exists an efficient non-discrete-logarithm bijective mapping to positive integers up to the group order. Since they cite the previous work by Kim et al. for this group definition, they are presumably referring to a Chaum group.

$t_{2,i} \neq \bar{t}_{2,i}$. In the honest case, the verifier's value expands as follows:

$$\bar{t}_{2,i} = \widetilde{y_2}^{y^{s_i}} = (g^{y^x})^{y^{s_i}} = g^{y^x \times y^{s_i} \pmod q}$$

Under normal circumstances, the expansion would continue:

$$g^{y^x \times y^{s_i} \pmod q} = g^{y^{x+s_i}} = g^{y^{x+\alpha_i - x}} = g^{y^{\alpha_i}}$$

However, $y^x \times y^{s_i} \pmod q \neq y^{x+s_i}$ in a Chaum group because $y^x \times y^{s_i} \pmod q \neq y^x \cdot y^{s_i}$. Consequently, the verification will fail and the protocol is incorrect.

Since Chaum groups cannot be used, the DDL-based BRAKEM constructions in this chapter use the "double-decker" group construction defined in Section 8.2.1, and a CG-DLEQ proof to move between groups.

8.3 Implementing BRAKEM$^{\text{DDL}}_{\star}$

Section 8.1 presented multiple BRAKEM constructions—with the most notable construction being BRAKEM$^{\text{DDL}}_{\star}$ as described in Section 8.2.8—in the form of abstract cryptographic protocols. Unfortunately, protocols that are defined in an abstract form in the literature often leave developers stranded, requiring them to figure out all of the implementation details. This is a reasonable practice, since particular protocol constructions come into and fall out of favor over time (e.g., group settings, primitive algorithms, and parameter sizes), and some cryptographic work is mainly intended to be an existence proof or to lay the groundwork for future protocols. However, protocols like Safehouse (and therefore BRAKEM$^{\text{DDL}}_{\star}$) are intended to be used in real-world deployments. These types of protocols can benefit immensely from proof-of-concept implementations that are developed alongside the design of the protocol. Results from the implementation can feed back into the design in order to improve it—a notable example is the ostensibly redundant presence of "burner keys" in BRAKEM$^{\text{DDL}}_{\star}$, which result in a large performance improvement in practice due to the real-world tradeoffs between different types of modular exponentiation algorithms. Details like these often cannot be discovered until a scheme has been implemented, and yet solving them requires changing the abstract protocol. Additionally, a working prototype ensures that the protocol design does not hide factors that would prevent real-world deployments, such as unreasonably large constants in time or space complexities.

This section describes important details concerning the practical deployment of BRAKEM$^{\text{DDL}}_{\star}$. Section 8.3.1 describes how to generate the group parameters required by BRAKEM$^{\text{DDL}}_{\star}$, and Section 8.3.2 discusses performance optimizations that are used by the prototype implementation.

8.3.1 Generating Group Parameters

In order to instantiate $\text{BRAKEM}_{\star}^{\text{DDL}}$, one must first set up the groups described in Section 8.2.1. Recall that these groups require a set of parameters with a very specific structure:

$$\begin{aligned} \log_2(p_3) &\approx 256 \\ \log_2(p_2) &\approx 3072 \\ p_2 &= 2qp_3 + 1 \\ p_1 &= 2p_2 + 1 \\ p_0 &= 2p_1 + 1 \text{ (optional)} \\ p_3, p_2, p_1, p_0 &\text{ are all prime} \end{aligned}$$

Note that p_0 is not required for $\text{BRAKEM}_{\star}^{\text{DDL}}$—it is only needed for $\text{BRAKEM}_{\star}^{\text{2DDL}}$. There are two important aspects to consider when selecting these parameters: the parameters must be valid, and it must be computationally feasible to find them. Section 8.3.1.1 explains what goes wrong if the parameters are invalid or maliciously chosen, and what steps can be performed to validate parameters supplied by others. Section 8.3.1.2 presents an efficient algorithm to compute suitable parameters, which was used in the implementation of the prototype. Section 8.3.1.3 describes techniques to ensure that the burner key x in $\text{BRAKEM}_{x}^{\text{DDL}}$ is used securely. Finally, Section 8.3.1.4 provides sample parameters.

8.3.1.1 Parameter Validation

The security of $\text{BRAKEM}_{\star}^{\text{DDL}}$, which is proven in Section 8.4, depends on the DDH problem being hard in $\mathbb{G}_2$ and the DL problem being hard in $\mathbb{G}_1$. The protocol defines these groups to be subgroups of $\mathbb{Z}_{p_2}^*$ (resp. $\mathbb{Z}_{p_1}^*$) generated by g_2 (resp. g_1) with prime order p_3 (resp. p_2). There are several pitfalls that can undermine the security of DDH in practice: [DCE17; VAS+17]

- **Wrong generator order**: If the generator generates the full multiplicative group (rather than a subgroup), then the DDH problem is easy to solve. For example, if g_1 has order $2p_2$ instead of p_2, then $\mathbb{G}_1 = \mathbb{Z}_{p_1}^*$ instead of an order p_2 subgroup of $\mathbb{Z}_{p_1}^*$. In this case, an attacker can learn a bit of the discrete logarithm by testing to see if an element is a quadratic residue.
- **Small group order**: When the order of the group is small, both Pollard's rho algorithm [Pol75] and the baby-step giant-step algorithm [Sha71] can efficiently solve the DL problem.

- **Composite group order**: If the order of the group is a smooth composite number with a known prime factorization, then the Pohlig-Hellman algorithm [PH78] can efficiently solve the DL problem.
- **Short exponents**: If the exponent for a particular DL problem instance lies within a known small range, Pollard's kangaroo algorithm [Pol00] can efficiently solve the problem.
- **Small modulus**: If the modulus for the group is small, then index calculus methods like the General Number Field Sieve (GNFS) can efficiently solve the DL problem. It is important to note that the majority of the computation for the attack depends only on the group definition, and not the particular group elements in a DL instance. This allows an attacker to perform a large precomputation for a specific modulus and then subsequently solve the DL problem very quickly for elements of the group as many times as desired [ABD+15].
- **Composite modulus trapdoor**: If the modulus for the group is composite, then it is possible for the parameters to be chosen in such a way that a trapdoor for the DL problem exists [DCE17]. An attacker with possession of the trapdoor can efficiently solve the DL problem, but the problem still remains hard for those without access to the trapdoor.
- **Small subgroup attacks**: If a participant can be tricked into performing group operations in a small subgroup (i.e., not the intended subgroup), then this can undermine the security of the scheme. For example, if the DH shared secret for an ElGamal ciphertext in $\text{BRAKEM}_{\star}^{\text{DDL}}$ is drawn from a small subgroup, the attacker can perform a small group order attack to recover the plaintext. Attackers can attempt a *small subgroup confinement* attack by sending elements outside of the intended subgroup to victims. A *small subgroup key recovery* attack can occur if the victim exponentiates elements from several small subgroups with the same secret exponent—an attacker can compute the discrete logarithm in each small subgroup and then use the Chinese remainder theorem to reconstruct the full secret. If the sender did not check for it, then this attack could occur in $\text{BRAKEM}_{\star}^{\text{DDL}}$.Encapsulate if the receiver public keys for a given ring were elements of distinct small subgroups.
- **Number field sieve backdoor**: If the group parameters are chosen maliciously, then it is possible to construct a prime modulus that enables an accelerated Special Number Field Sieve (SNFS) attack against the DL problem when given access to the trapdoor relationship between two polynomials [FGHT17]. Moreover, it is feasible to hide this special prime form. Asymptotically, the attack reduces the security of the DL problem from $O(2^{\lambda})$ to $O(2^{\lambda/2})$ against an attacker armed with the backdoor, but the sizes used in practice are too small for the asymptotic bounds to be accurate [FGHT17, §6].

- **Long-term key reuse across security contexts**: If an attacker compromises a DH public key by solving the DL problem, they will be able to decrypt any ElGamal ciphertexts encrypted for that key. In the context of $\text{BRAKEM}_x^{\text{DDL}}$, compromising the burner key x would undermine all keys encapsulated using the scheme.

In order to prevent all of these attacks, the following constraints must be satisfied:[13]

- **Correct generator orders**: g_2 must have order p_3, g_1 must have order p_2, and g_0 must have order p_1. This condition can be validated as follows:
  - the required structure of (q, p_3, p_2, p_1, p_0) holds;
  - $g_2 \neq 1$; $g_1 \neq 1$; $g_0 \neq 1$;
  - ${g_2}^{p_3} = 1 \pmod{p_2}$; ${g_1}^{p_2} = 1 \pmod{p_1}$; and ${g_0}^{p_1} = 1 \pmod{p_0}$.

  This condition prevents attacks against DDH due to wrong generator orders.

- **Large group order**: $\log_2(p_3) \approx 256$. This provides a 128-bit security level against Pollard's rho algorithm and the baby-step giant-step algorithm. Given the required relationship between the parameters, this automatically implies that $\mathbb{G}_1$ and $\mathbb{G}_0$ have extremely large orders that also prevent this attack.

- **Prime group order**: If p_3, p_2, and p_1 are all prime, then the orders of $\mathbb{G}_2$, $\mathbb{G}_1$, and $\mathbb{G}_0$ are all prime. This prevents the use of the Pohlig-Hellman algorithm.

- **Full-range exponents**: In any security proof where the DL problem must be hard, the exponents for the group elements in question must be selected uniformly at random from the whole range up to the group order. This applies mainly to the private keys produced by $\text{BRAKEM}_\star^{\text{DDL}}$.KeyGen and the ElGamal randomness generated by $\text{BRAKEM}_\star^{\text{DDL}}$.Encapsulate. This condition provides at least a 128-bit security level against Pollard's kangaroo algorithm.

- **Large moduli**: $\log_2(p_2) \approx 3072$. This is commonly believed to provide a 128-bit security level against index calculus methods for solving the DL problem. Given the required relationship between the parameters, this automatically implies that p_1 and p_0 are also large enough to prevent the attack in their respective groups. The size of p_2 is also believed to be large enough to prevent GNFS precomputations in practice, even if all of the attacker's resources are dedicated to a single p_2 reused across many security contexts [ABD+15]. Additionally,

[13] Constraints involving p_0 and g_0 are only necessary for $\text{BRAKEM}_\star^{\text{2DDL}}$ and not for $\text{BRAKEM}_\star^{\text{DDL}}$. They are included here for completeness.

this size of p_2 is resistant to SNFS backdoors in the event that p_2 was chosen maliciously, since the acceleration provided by the backdoor is insufficient to solve the DL problem in practice (although $\mathbb{G}_2$ would not provide the claimed 128-bit security level against such an attacker). The general problem of preventing backdoors (including but not limited to SNFS backdoors) from being hidden in the parameters is revisited later in this section.

- **Prime moduli**: If p_2, p_1, and p_0 are all prime, then $\mathbb{G}_2$, $\mathbb{G}_1$, and $\mathbb{G}_0$ all have prime moduli. This prevents the presence of a backdoor based on a maliciously chosen composite modulus.

- **Prime q**: If p_3, p_2, and q are all prime and $p_2 = 2qp_3 + 1$, then all of the subgroups in $\mathbb{G}_2$ have an order $\geq p_3$, with the lone exception of the subgroup $\{1, p_2 - 1\}$ of order 2. This means that $\mathbb{G}_2$ contains no small subgroups in which to conduct small subgroup confinement or key recovery attacks, aside from the required subgroup of order 2.[14] If p_1 is prime, $p_1 = 2p_2 + 1$, and $p_0 = 2p_1 + 1$, then the same is true for $\mathbb{G}_1$ and $\mathbb{G}_0$.

- **Group membership tests**: When receiving alleged group elements, the receiver must verify that they belong to the subgroup with the expected order (and in particular, not to the subgroup of order 2). When combined with the previous condition, these checks fully prevent small subgroup attacks.

- **Secure burner keys**: Although the other conditions ensure that the DL problem is considered too difficult to solve for even one instance, reusing the same burner key x in $\text{BRAKEM}_x^{\text{DDL}}$ for all deployments would still present an unnervingly large and lucrative attack target. At a minimum, each deployment should use a different burner key. A new burner key can also be generated more frequently (e.g., for each communication context, or each time epoch). In the extreme case, the burner key can be omitted from the scheme entirely, with $\text{BRAKEM}_x^{\text{DDL}}$ adjusted accordingly to perform the Π_0 NIZKPK for the first true recipients from each ring. When a burner key is used, precautions must be taken to ensure that adversaries cannot learn the discrete logarithm without solving the DL problem itself; Section 8.3.1.3 discusses a method to accomplish this.

All of the parameter validation steps mentioned above are trivial to verify, with the exception of the constraints that p_3, p_2, p_1, p_0, and q must be prime. How can a verifier check that a given parameter is prime? Trial division becomes intractable long before reaching numbers as large as p_3 (which itself is astronomically smaller than q), so it is not an option. An efficient alternative is a probabilistic primality testing algorithm such as the Fermat or Miller-Rabin [Mil76] tests.

[14] Technically, the minimal condition to prevent these attacks is that q contains no prime factors smaller than p_3. For simplicity and because there is no good reason to do otherwise, the group parameters in this chapter use a prime q.

However, these tests are sound but not complete: they either conclusively identify that a number is composite, or return an inconclusive result. Consequently, the tests are typically executed multiple times (with fresh randomness for each execution) in order to decrease the probability that they fail to identify a composite number. In an adversarial settings, maliciously chosen composite numbers can cause the accuracy of the tests to degrade to the worst-case scenario [AMPS18; GMP19]. This means that these tests must be run enough times to become confident that an "inconclusive" results indicates primality when accounting for worst-case accuracy.[15] The Baillie-PSW test [PSW80] is more robust: there are no currently known Baillie-PSW pseudoprimes (i.e., composite numbers that pass the test). While it is conjectured that there are no Baillie-PSW pseudoprimes for numbers less than a few thousand bits, it is also known that there are infinitely many such pseudoprimes [Pom84]. In practice, performing the Baillie-PSW test or sufficiently many iterations of the Miller-Rabin test provides sufficient confidence of primality in adversarial settings. However, it is possible to do better: with modern tools, it is possible to verify that a cryptographically large number is prime with *certainty*.

In contrast to probabilistic primality tests, there are various methods to *prove* that a number is prime [Ber04]. These methods output a *primality certificate* that contains values used as input to a theorem about prime numbers. A verifier can use the contents of a primality certificate and the associated theorems to efficiently show that a number is prime.

There are two main techniques that are used to generate primality certificates in practice: Pocklington's theorem [Poc1914], and the Elliptic Curve Primality Proving (ECPP) algorithm [AM93; Mor98]. Pocklington's theorem can be used in a recursive primality proving algorithm. For a given candidate N, the algorithm must find a large prime factor ρ of $N-1$ such that $\rho \geq \sqrt{N}$ and ρ satisfies some other criteria. The factor ρ is then proven to be prime recursively. Unfortunately, this algorithm does not work for all primes: if at any point it encounters an input N such that *all* of the prime factors of $N-1$ are $< \sqrt{N}$, then the theorem cannot be applied. There is a generalized form of Pocklington's theorem that does not require a single large prime factor ρ. Instead, the generalized version can work with any partial prime factorization $A = \prod_{i=1}^{m} {\rho_i}^{e_i}$ such that $A \geq \sqrt{N}$ and $AB = N-1$ for some integer B, which need not be factored. This version of the theorem is more flexible, but still requires the algorithm to find a large prime factor of $N-1$ if one exists (otherwise A will not be large enough). The SafeCurves website [BL14] provides primality certificates for various standard elliptic curve parameters using the generalized form of Pocklington's theorem exclusively. The ECPP algorithm is also recursive, but it is far more efficient than Pocklington's theorem in practice. Efficiently executing the algorithm requires constructing elliptic curves with a given number of points using complex multiplication, where the number of points is a multiple of a smaller prime. A proof certificate

[15] In the worst-case scenario (which occurs in adversarial settings), the Miller-Rabin test must be run 64 times in order to declare a number to be prime with an error probability of 2^{-128} [Mil76].

for the smaller prime is generated recursively. Unlike Pocklington's theorem, the ECPP method is able to produce primality certificates even when a candidate N has large prime factors in $N - 1$ that cannot be found efficiently (e.g., in cases where $N - 1$ is an RSA number).

When verifying group parameters for BRAKEM$^{\text{DDL}}_{\star}$, verifiers should require primality certificates for p_3, p_2, p_1, p_0, and the factors of q; equivalently, a verifier may merely require that the certificates can be efficiently generated at verification time. The closed-source Primo software [Mar15] generates and verifies primality certificates using Pocklington's theorem (in the non-generalized form) and ECPP. The Primo certificate format is supported by several open-source software projects,[16] making it the most widely supported primality certificate format currently in use.

One final challenge remains: even if all of the aforementioned constraints are verified, this does not guarantee that the parameters are free from backdoors. For example, parameters that are generated using the hidden SNFS technique [FGHT17] cannot be detected. While the SNFS acceleration may not be enough to break the 3072-bit prime modulus p_2 in practice, it would certainly be better to prevent this acceleration. Moreover, the SNFS backdoor is illustrative of a more general problem: if a malicious parameter generator is aware of an attack against DL (or DDH) that only applies to a subset of possible parameters, then it can choose parameters that enable its (potentially trapdoor-based and undetectable) attack. To mitigate this weakness, either the parameters must come from a trusted parameter generator, or the parameter generator's flexibility when choosing the parameters must be sufficiently constrained to make it very unlikely that the parameters are vulnerable to their attack. There are several common ways to accomplish this restriction [BCC+15]:

- The generator may be required to provide a "seed" that, when input into a cryptographic hash function, outputs an uncontrollable value. The output of the hash is translated into the structured parameters according to a defined algorithm.
- The generator may be required to construct the parameters using a "nothing-up-my-sleeve" number such as π or e. For example, the prime numbers for the scheme might be chosen based on some initial digits of π plus some small value ϵ. Alternatively, a "nothing-up-my-sleeve" number might be chosen as the seed for the hash-based approach.
- The generator may be required to select the "minimal" parameters. For example, the prime numbers for the scheme might be the smallest possible primes that satisfy the requirements.

[16] PARI/GP (`https://pari.math.u-bordeaux.fr/`) can generate Primo certificates, but not verify them. CoqPrime (`https://github.com/thery/coqprime`) and `ecpp-verifier` (`https://github.com/tomato42/ecpp-verifier`) can verify Primo certificates, but not generate them.

All of these approaches are still subject to manipulation to varying degrees. For example, if the parameters are required to be the output of a hash function, a malicious parameter generator could try many seeds and hash functions until they find one that generates parameters subject to their attack. Similarly, an adversary can try many plausible "nothing-up-my-sleeve" numbers and encoding procedures until they generate vulnerable parameters. Of the three approaches, restricting the parameter generator to choosing the "minimal" parameters removes the most, but not all, adversarial flexibility [BCC+15, §6]. The remaining flexibility includes the prime size and the prime "shape" (e.g., "ordinary" primes, pseudo-Mersenne primes, "Montgomery-friendly" primes, among others).

To limit the likelihood that a malicious parameter generator can generate weak $\text{BRAKEM}_{\star}^{\text{DDL}}$ parameters, the following constraints should be verified:

- p_3 should be equal to the order of a standardized elliptic curve that is suitable for ElGamal encryption (i.e., DDH should be hard). Translation from $\mathbb{G}_2$ into an elliptic curve group is used by Safehouse to accelerate certain features. When the groups have the same order, it is possible to efficiently prove the correctness of this translation using a DLEQ NIZKPK. This requirement is not necessary for general use of $\text{BRAKEM}_{\star}^{\text{DDL}}$.
- q should be the smallest prime such that the other constraints are satisfied.
- g_1 and g_0 should be the smallest values strictly greater than 1 with the correct order: ${g_1}^{p_2} = 1 \pmod{p_1}$ and ${g_0}^{p_1} = 1 \pmod{p_0}$.
- g_2 should be $z^{2q} \pmod{p_2}$ for the smallest z such that $z^{2q} \neq 1 \pmod{p_2}$. This ensures that g_2 will be in the subgroup of order p_3.

A malicious parameter generator's only flexibility given these constraints is the selection of the elliptic curve group, which determines p_3. Of course, the list of constraints presented in this section are themselves a flexibility: they are only one of many possible "reasonable" methods for parameterizing $\text{BRAKEM}_{\star}^{\text{DDL}}$. Ultimately, this is an inherent limitation of trust in parameter generation. For this particular set of parameters, the set of possible "reasonable" constraints is likely small enough that a malicious parameter generator would require an attack that is applicable to a very large proportion of typical finite-field DH setups.

Note that $\log_2(q) = \log_2(p_2) - \log_2(p_3) - 1$ by definition. When q is a prime this large, $q - 1$ is likely to contain large prime factors that preclude the use of Pocklington's algorithm to prove that q, p_2, p_1, or p_0 are prime. Luckily, ECPP is still efficient enough to produce primality certificates for primes of this size.

8.3.1.2 Finding Primes

Section 8.3.1.1 described the steps required to verify BRAKEM$^{\text{DDL}}_{\star}$ group parameters. However, it remains to be shown how to find suitable parameters in the first place. With careful consideration, finding appropriate parameters can be done efficiently. The performance of the parameter generation algorithm is particularly important for BRAKEM$^{\text{2DDL}}_{\star}$, which requires the extra prime p_0—the time required to locate appropriate values is exponential in the number of primes that must be found. A parameter generation program was developed as part of this work. With the algorithm described in this section, the generation program is able to find appropriate parameters for BRAKEM$^{\text{2DDL}}_{\star}$ within hours using an academic-grade computing cluster. In contrast, finding parameters for BRAKEM$^{\text{DDL}}_{\star}$ on a consumer-grade desktop computer using the algorithm can usually be done within several minutes. The large size of q is a significant contributing factor to the runtime cost of the algorithm.

Recall that the following equations must be satisfied, where all variables are prime:

$$\begin{aligned} \log_2(p_3) &\approx 256 \\ \log_2(p_2) &\approx 3072 \\ p_2 &= 2qp_3 + 1 \\ p_1 &= 2p_2 + 1 \\ p_0 &= 2p_1 + 1 \text{ (optional)} \end{aligned}$$

As noted in the previous section, p_3 is provided as an input to the parameter generation algorithm in order to accelerate certain features of Safehouse.

An efficient way to generate the remaining parameters is to generate a prime q with the correct number of bits. Note that p_3 and q fully determine the other parameters. After generating a prime q, the other parameters can be tested using the Fermat and Baillie-PSW [PSW80] tests to see if they are prime. The algorithm terminates when all parameters are prime. The Fermat primality test is much more efficient than the Baillie-PSW test, so it should be run first in order to speed up elimination of obviously composite candidates.[17]

Blindly generating q at random and running the primality test on the resulting parameters is too inefficient. To make this process practical, an extra layer must be added to quickly eliminate many candidates before they reach the primality tests. This is where prime number sieves are useful. When searching for primes of this size using modern processor architectures, it is

[17] The Baillie-PSW begins with a single application of the Miller-Rabin test, which essentially includes a Fermat test. However, since the algorithm searches for multiple primes simultaneously, it is more efficient to test all of the candidates with a Fermat test before incurring the cost of even a single full Baillie-PSW test.

Algorithm 8.1 A SEGMENTED PRIME SIEVE. (Refs: 213 [a] [b])

Subroutine | Segmented Prime Sieve in $[n_1, n_2]$

Find all primes in $[2, k)$, $\mathbf{P}$, using a prime sieve.
Initialize A, an array of $n_2 - n_1 + 1$ bits initially set to 1.
for each $(p' \in \mathbf{P})$ {
  for each (integer m such that $mp' \in [n_1, n_2]$) {
    $A[mp'] \leftarrow 0$.
  }
}
for each $(A[i]$ such that $A[i] = 1)$ {
  if (IsPrime$(A[i])$) Output $A[i]$.
}

best to use a segmented sieve [BH77]. The general structure of a segmented prime sieve is shown in Algorithm 8.1. The preprocessing step of finding all primes in $[2, k)$ for some k can be implemented using a traditional prime sieve. After finding these "small" primes, the data can be reused to eliminate candidates from multiple segments in parallel. Each processor core i can search its own segment $[n_{i,1}, n_{i,2}]$ with the desired bit length for the prime. After preprocessing is complete, the sieve becomes a very efficient mechanism for eliminating candidates before having to run the comparatively expensive primality tests.

While the combination of a segmented sieve with a primality test is already quite efficient, a significant performance improvement can be achieved by using a *wheel*. This technique was first described in detail by Pritchard [Pri82]. It begins with the observation that sieves like the segmented sieve in Algorithm 8.1 do not need to consider even numbers, which cannot be prime (aside from 2, which is typically outside the search range). Consequently, the array A only needs to store half of the bits, and mp' can be ignored when m is a multiple of 2. However, there is no need to stop at 2: multiples of 3 can also be excluded from the memory layout and candidate elimination passes. When combined with a segmented prime sieve, the resulting construction is called a segmented wheel sieve [Pri83]. In general, a "factor base" of primes $\mathbf{F} = \{f_1, f_2, \ldots, f_m\}$ eliminates a pattern of prime candidates that repeats after every multiple of $h = \prod_{1 \le i \le m} f_i$. This can be visualized as a "wheel" with segments corresponding to $[1, h]$ on its circumference, where segments are "spokes" if and only if they correspond to a number that is coprime to every element of the factor base. As this wheel "rolls" along the number line, all integers that did not contact a "spoke" are eliminated from the sieve's memory layout. Figure 8.1 depicts a wheel with factor base $\{2, 3, 5\}$ on the number line.

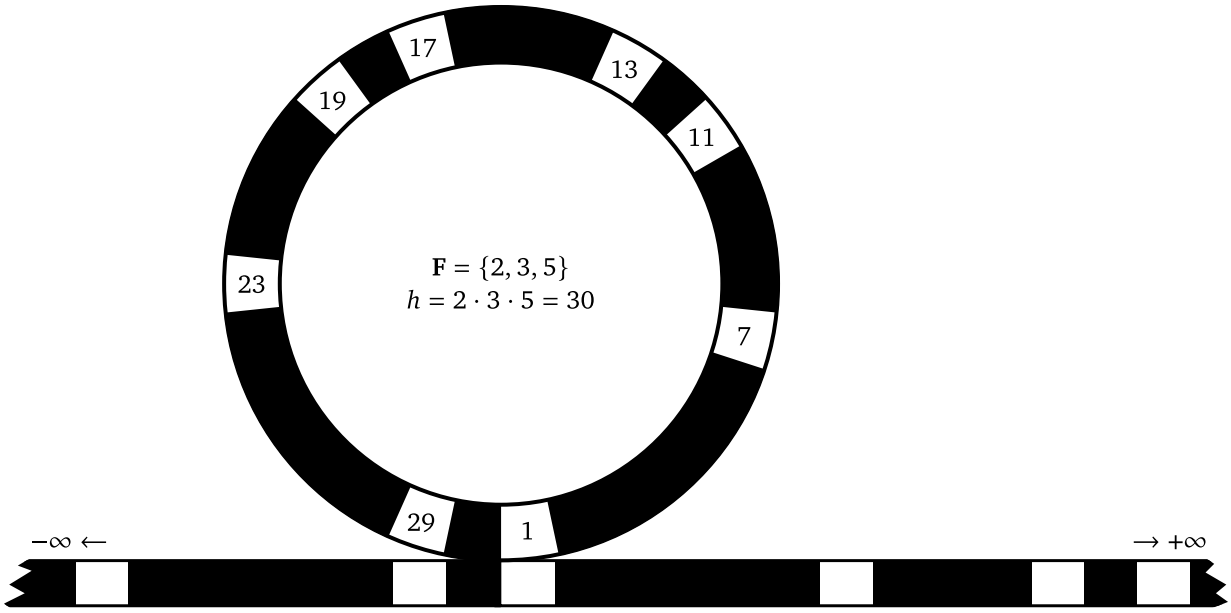


Figure 8.1 A SMALL PRIME SIEVING WHEEL ON THE NUMBER LINE. The wheel's factor base is $\{2, 3, 5\}$. Filled segments are provably composite and are not represented in the prime sieve's memory layout. (Refs: 213, 215, and 216)

The location of spokes in a wheel and the distance between them can be precomputed, making it easy to look up the next spoke position and index given a value modulo h. As an additional optimization, the initial Fermat primality test can be modified to use h as the base. This ensures that $\gcd(h, n) = 1$ for a candidate n, since n has already survived the sieve at the time of the test; if n is not a Carmichael number, then the test will eliminate a composite n half of the time. Several aspects of the segmented wheel sieve are also simplified by ensuring that segments always start and end on multiples of h.

An application-specific optimization performed by the $\text{BRAKEM}_\star^{\text{DDL}}$ parameter generator program is to make the segmented wheel sieve aware of the required prime relationships, eliminating even more candidates during the relatively inexpensive sieving process. Consider the small prime $p' \in \mathbf{P}$ and a candidate n. Normally, the sieve eliminates n if $n = 0 \pmod{p'}$. However, when the overall algorithm is sieving for q using a segmented wheel sieve, there are other applicable constraints:

$$\begin{aligned} q &\neq 0 \pmod{p'} \\ p_2 = 2qp_3 + 1 &\neq 0 \pmod{p'} \\ p_1 = 2(2qp_3 + 1) + 1 = 4qp_3 + 3 &\neq 0 \pmod{p'} \\ p_0 = 2(2(2qp_3 + 1) + 1) + 1 = 8qp_3 + 7 &\neq 0 \pmod{p'} \text{ (optional)} \end{aligned}$$

During the sieving step with each of the small primes, three (for $\text{BRAKEM}_\star^{\text{DDL}}$) or four (for $\text{BRAKEM}_\star^{\text{2DDL}}$) passes through the segment are performed. Each pass eliminates candidates that would cause one of $\{q, p_2, p_1, p_0\}$ to be divisible by p'. Since p_3 is given as an input to the program, an initial preprocessing step can cache the constraints for each small prime p':

$$\begin{aligned} q &\neq 0 \pmod{p'} \\ q &\neq (-2p_3)^{-1} \pmod{p'} \\ q &\neq -3 \cdot (4p_3)^{-1} \pmod{p'} \\ q &\neq -7 \cdot (8p_3)^{-1} \pmod{p'} \text{ (optional)} \end{aligned}$$

This technique is an extended form of an approach that can be used to find safe primes [Wie03].

When implementing a segmented wheel sieve, it is important to take advantage of the wheel structure. When sieving with a small prime p' in segment $[n_1, n_2]$, it is possible to quickly compute all multiples of p' that lie on a spoke within the segment—in Figure 8.1, these correspond to unfilled values on the number line that are also a multiple of p'. These are the values that must be eliminated during the sieve. An important observation is that any multiple mp' of p' lies on a wheel spoke if and only if m lies on a wheel spoke (p' always lies on a wheel spoke

because by definition it cannot be divisible by any prime in the factor base). To begin the sieving process for p', the initial multiplier $m_0 = \lfloor {}^{n_1}/_{p'} \rfloor$ is computed. The wheel is then consulted to find the subsequent spokes $m_1, m_2, \ldots$, where $m_1 = m_0$ if m_0 itself is already on a spoke. The sequence $m_1 p', m_2 p', \ldots$ will contain all of the candidates to be eliminated by the sieve. The loop can terminate for p' as soon as it encounters an $m_i p' > n_2$. Another optimization is to only store ${}^{n_2 - n_1 + 1}/_h$ bits when sieving a segment; efficient mapping from a candidate n to a memory position can be done by computing the number of wheel rotations ${}^{n - n_1}/_h$ and the spoke index of $n - n_1 \pmod h$, then mapping these two values to a bit and byte position.

When performing a sieving step with a prime p' and a constraint $q \neq r \pmod{p'}$, the aforementioned procedure is performed as described if $r = 0$. For the three constraints where $r \neq 0$, the procedure must be slightly adjusted. Rather than finding multipliers $m_1, m_2, \ldots$ on wheel spokes in the usual way, the wheel is first "rotated" so that the spokes are offset by $r' = {}^{r}/_{p'} \pmod h$. For example, the spoke at position 11 in Figure 8.1 would move to position 3 if $r' = 8$. This solution works because if a candidate q is equal to the remainder $r \pmod{p'}$ and q also falls on a wheel spoke in the sieving segment, then:

$$
\begin{aligned}
& q &&= r \pmod{p'} && \triangleright \text{ Formula for a constraint violation} \\
& q &&= mp' + r && \triangleright \text{ Rewrite mod in terms of a multiple of } p' \\
\forall_{f \in \mathbf{F}} \quad & mp' + r &&\neq 0 \pmod f && \triangleright \ q \text{ must lie on a spoke} \\
\forall_{f \in \mathbf{F}} \quad & m + r \cdot p'^{-1} &&\neq 0 \pmod f && \triangleright \text{ Multiply by } p'^{-1} \\
\forall_{f \in \mathbf{F}} \quad & m + r' &&\neq 0 \pmod f && \triangleright \text{ Definition of } r'
\end{aligned}
$$

The inverse of p' modulo f for any $f \in \mathbf{F}$ is guaranteed to exist because p' is coprime to f. The final constraint that $\forall_{f \in \mathbf{F}}\ m + r' \neq 0 \pmod f$ is exactly the definition of finding a value $m' = m + r'$ on a wheel spoke modulo h; q can then be computed as $q = (m' - r') \cdot p' + r$. Alternatively, the constraint can be viewed as finding m on the wheel after it has been rotated by r'. This means that candidates satisfying $q = r \pmod{p'}$ and $\forall_{f \in \mathbf{F}}\ q \neq 0 \pmod f$ can be found by finding multipliers $m_1, m_2, \ldots$ that fall on spokes in the rotated wheel. The candidates to be eliminated by the sieve will be $m_1 p' + r$, $m_2 p' + r$, and so on. Note that the value of r' depends only on r and p', so it can be precomputed for all constraints and reused across all sieves.

Once appropriate prime parameters have been discovered, the generators $g_1 \in \mathbb{G}_1$ and $g_0 \in \mathbb{G}_0$ can be found with a simple linear search checking candidates z for $z^{p_2} = 1 \pmod{p_1}$ or $z^{p_1} = 1 \pmod{p_0}$. This search will terminate quickly because the cofactor is very small (the cofactor for both groups is 2). A similar approach does not work for finding a generator $g_2 \in \mathbb{G}_2$ because of its large cofactor. Instead, a generator can be found by choosing a $z \in [2, p_2)$ and eliminating the component from the cofactor subgroup: $g_2 = z^{2q} \pmod{p_2}$.

The overall approach used by the parameter generator is summarized in Algorithm 8.2. The parameters produced by this algorithm pass all of the validation tests described in Section 8.3.1.1.

Algorithm 8.2 THE GROUP PARAMETER GENERATOR ALGORITHM. (Ref: 216)

Subroutine | ParameterGenerator$(p_3) \rightarrow (p_3, q, p_2, p_1, p_0, g_2, g_1, g_0, \pi)$

Input requirement: p_3 is a prime and $\log_2(p_3) \approx 256$.

Find all primes in $[2, k)$, **P**, using a prime sieve.
Compute the four constraints for each prime in **P** given p_3.
while (true) {
  Let $\ell = 3072 - \log_2(p_3) - 1$.
  Perform a parallel segmented wheel prime sieve in $[2^\ell, 2^{\ell+1})$ with the constraints.
  for each (candidate q that survives the sieving process) {
    $p_2 \leftarrow 2qp_3 + 1$
    $p_1 \leftarrow 2p_2 + 1$
    $p_0 \leftarrow 2p_1 + 1$
    if (**!**FermatPrime(q, h) || **!**FermatPrime(p_2, h)) **continue**
    if (**!**FermatPrime(p_1, h) || **!**FermatPrime(p_0, h)) **continue**
    if (**!**BailliePSW(q) || **!**BailliePSW(p_2)) **continue**
    if (**!**BailliePSW(p_1) || **!**BailliePSW(p_0)) **continue**
    Find smallest $g_1 > 1$ such that ${g_1}^{p_2} = 0 \pmod{p_1}$.
    Find smallest $g_0 > 1$ such that ${g_0}^{p_1} = 0 \pmod{p_0}$.
    Find smallest $z > 1$ such that $z^{2q} \neq 1 \pmod{p_2}$.
    $g_2 \leftarrow z^{2q}$.
    Produce primality proofs π for p_3, q, p_2, p_1, and p_0 using the ECPP algorithm.
    return $(p_3, q, p_2, p_1, p_0, g_2, g_1, g_0, \pi)$.
  }
}

However, care must be taken to ensure that the parallelized segmented sieve scans the segments in sequential order so that the resulting q is minimal. If a solution is found in a greater segment before a lesser segment has been completely processed, the parameter generator should not output the solution until all unfinished segments have been scanned; this ensures that the minimal solution is returned.

There are several parameters to tune for optimal performance in the parameter generator algorithm. Larger values of k eliminate more candidates during the sieve, but consume more memory, increase the sieving time, suffer from diminishing returns, and harm memory locality. Increasing the size of the factor base **F** eliminates more candidates, but it increases complexity, harms cache locality in several tight loops, and is subject to diminishing returns. Increasing

the segment length consumes more memory and increases latency, but it reduces the overall cost of the sieving process due to reducing the frequency at which the small primes need to be "aligned" with the segment—computing the initial multiplier m_0 is expensive for numbers as large as q. The parameter generation program developed as part of this work uses $k = 2^{18}$, $\mathbf{F} = \{2, 3, 5, 7, 11\}$, $h = 2310$, and a segment length of 99 999 900. While this exceeded the capacity of the processor's L2 cache, there was a net benefit to performance due to starting new sieves less frequently.

8.3.1.3 Burner Key Security

The presence of the burner key x in $\text{BRAKEM}_x^{\text{DDL}}$ is an important optimization that improves the performance of Π_0. However, it can also become a security weakness if it is not handled properly: if an adversary learns $\text{dlog}_{g_2}(x)$, then the confidentiality of the scheme is compromised until the burner key changes.

To avoid this problem, x should be set to an element of $\mathbb{G}_2$ chosen uniformly at random. As long as the adversary cannot force x to equal an element for which they know the discrete logarithm, then they cannot break the scheme by attacking x without solving the DL problem in $\mathbb{G}_2$. An easy way to prevent an adversary from assigning x to a chosen value is to derive it from the output of a cryptographic hash function. Even if the adversary has control over the input to the hash function, its preimage resistance prevents them from choosing an input that would produce their desired element.

Given a cryptographic hash function H that maps strings to values in $\mathbb{Z}_{p_2}^*$, the recommended scheme for choosing x is to use H with a common reference string s as input and then mapping the output into the correct subgroup. The string s does not need to be random, but it must be known by all participants in the scheme and it should capture the "security context" of the conversation. For example, s might be an identifier for the conversation, or the name of the service that has deployed $\text{BRAKEM}_x^{\text{DDL}}$. The participants can derive x from s in order to instantiate $\text{BRAKEM}_x^{\text{DDL}}$. To map the hash output, which is in $\mathbb{Z}_{p_2}^*$, into the correct subgroup $\mathbb{G}_2$, the safest option is to eliminate the portion of the value that comes from the multiplicative subgroup with order $2q$ (i.e., the cofactor). In summary, x is computed from s as follows:

$$x = H(s)^{2q \cdot \left((2q)^{-1} \pmod{p_3}\right)} \pmod{p_2}$$

This function ensures that the distribution of x in $\mathbb{G}_2$ is uniformly random, and that an adversary cannot force x to equal a specific value, even with full control over s. Developers should take care to ensure that the distribution of the output of H is uniform. It is easy to implement H using an Extendable Output Function (XOF), such as SHAKE-128, using rejection sampling: generate

$\lceil \log_2(p_2) \rceil$ bits of output from XOF(s), interpret it as a number, and if the result is $\geq p_2$ then try again with subsequent output bits. Implementing H using a standard hash function H', such as SHA2-256, is similar but slightly trickier: H' can be called repeatedly with input $s \| c$ for an incrementing counter c in order to generate enough bits to perform rejection sampling. This transformation of a hash into a pseudorandom function is a simplified form of the HKDF scheme introduced by Krawczyk [Kra10].

8.3.1.4 Sample Parameters

The following sample parameters were found by the parameter generation program described in Section 8.3.1.2 when setting p_3 to the order of the Curve25519 [Ber06] elliptic curve group:[18]

$p_3 =$ `0x1000000000000000000000000000000014DEF9DEA2F79CD65812631A5CF5D3ED`

$q = 2^{2818} + 0x39E6AB2D1D$
$= 0x4000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`00000000000000000000000000000039E6AB2D1D`

$p_2 =$ `0x8000000000000000000000000000000A6F7CEF517BCE6B2C09318D2E7AE9F6800000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`0000000000000000000000000000000000000000000000000000000000000000`
`00000000000000000000000000000073CD565A3A0000000000000000000970E7D75D8CE35DCBD`
`E07B7E794BC016876D155B3`

[18] These values are intended to demonstrate that the algorithm is capable of producing a valid solution. Developers should always validate parameters using the techniques described in Section 8.3.1.1 when using parameters provided by others.

$p_1 =$ 0x10000000000000000000000000000000014DEF9DEA2F79CD65812631A5CF5D3ED00000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000E79AACB47400000000000000000000012E1CFAEBB19C6BB97
BC0F6FCF297802D0EDA2AB67

$p_0 =$ 0x20000000000000000000000000000000029BDF3BD45EF39ACB024C634B9EBA7DA00000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000001CF355968E800000000000000000000025C39F5D76338D772F
781EDF9E52F005A1DB4556CF

$g_2 = 2^{2q} \pmod{p_2}$
$=$ 0x6498BF4C8E3CBE74CDAF2CEC5B2023A1DC3F16BE4471ECD719BDD430DDF4AA995F6FEDE692BDDD8FC
563DB51F3C6F32F6A32E004C37F757264A64C07667A010B099F449FE894F2FF88FF37FF2E177CEED6EC
79FE27CF104E3F9A68136C835FF87418DDE6C8083B34B8DCF28B3C739D3103274A6B6C6716FD6CAD36C
0626FDD29A3449D8B1C883B0067D3A61DEADA4D74411C068B088F5C7AB140C67A0B53B4C08FDD578799
C79163DAB6895D76F7C06923053A859F33A7A409A379EBF5D7895B2EC749DBB46EC8A6B98CE2D20E858
D527E13727742EB9CE7305C5E3E88DF8F4C76264C2A703B0762F010ACA11B80F8A340D5C8BFB2EF3360
EDB36C0DFEC74F6D8BB34E20F5134286BF30258206A3ACFC37D0ABA3C751664CF9630A502476D1EC129
8F6C623EB61CD3FF94B7B61490200960D16D21A86EC2EF4AA2924B530B8BEAE997AA9934425CAFA9B2D
C8A9EB8FA4C22E8294BC4D867127BB83AC5834ABDD8360915E1AC2CF0742AB0CFDA287BC9E98F7B703E
A7112F1D52E930ABFAB21F6

$g_1 = 2$

$g_0 = 2$

8.3.2 Performance Optimization

The prototype implementation of $\text{BRAKEM}_{\star}^{\text{DDL}}$ that was developed as part of this work is intended to demonstrate a reasonable bound on the real-world performance of the system. As such, although the prototype does not use all of the techniques available to a production-ready library, it does employ many of the most important performance optimizations that would be used in practice. The prototype was written in the Go programming language,[19] since it provides a good mix of concurrency primitives, math and cryptography libraries, safety, the ability to include assembly (for targeted performance gains at the cost of safety), and an ecosystem that can support the higher-level features required by Safehouse.

Optimizing a $\text{BRAKEM}_{\star}^{\text{DDL}}$ implementation is important in order to achieve reasonable running times: since the system uses "finite-field" DH groups, it is significantly slower than the elliptic curve techniques that have become the norm for real-world cryptography. The size of the $\text{BRAKEM}_{\star}^{\text{DDL}}$.Encapsulate proofs can be marginally decreased by merging the Fiat-Shamir [FS87] challenges: by combining all of the random oracle inputs within the NIZKPKs into a single call and reusing the output, only one hash value needs to be transmitted. Aside from this standard optimization, there is no way to decrease the size of the encapsulations. However, the running time can be improved far beyond what a naïve implementation can accomplish.

Almost all of the running time for the $\text{BRAKEM}_{\star}^{\text{DDL}}$ functions is spent performing modular exponentiation in some form. Modular exponentiation is implemented as a sequence of modular multiplications. The length of this sequence is proportional to the bit length of the exponent. This is particularly expensive for any modular exponentiations in $\mathbb{G}_1$ since these involve 3072-bit exponents.

Aside from the naïve and intractable solution of computing v^y by multiplying $y - 1$ times, the most basic modular exponentiation algorithm is the "square-and-multiply" algorithm shown in Algorithm 8.3. This algorithm is easy to implement given an algorithm for modular multiplication with the required bit length. However, there are many ways to drastically outperform this basic algorithm; since modular exponentiation has widespread importance in cryptography, optimizing it has been the subject of many research efforts. Optimizations fall into two categories: speeding up the modular multiplication operation, and more efficient exponentiation algorithms. The largest speed improvements in the latter category come from taking advantage of special cases, such as when the base or exponent are known long in advance, allowing precomputation to be performed.

This section covers the optimizations employed by the $\text{BRAKEM}_{\star}^{\text{DDL}}$ prototype, based on the exact operations that need to be performed:

[19] ^ https://golang.org/

Algorithm 8.3 SQUARE-AND-MULTIPLY MODULAR EXPONENTIATION.
(Refs: 221, 227 a b, 228, 231, 234, and 281)

Subroutine | ModExp$(v, y, p) \rightarrow z$

Output: $z = v^y \pmod{p}$.

$z \leftarrow 1$.
for (bit b in y from most to least significant) {
  $z \leftarrow z \cdot z \pmod{p}$.
  if $(b = 1)$ {
    $z \leftarrow z \cdot v \pmod{p}$.
  }
}
return z.

1. Section 8.3.2.1 discusses how to speed up the modular multiplication function, which is needed by all exponentiation methods.

2. Section 8.3.2.2 discusses optimizing modular squaring, which is a special case of multiplication. Modular squaring constitutes the bulk of the time spent for most modular exponentiations.

3. Section 8.3.2.3 discusses a faster technique than square-and-multiply for "ordinary" modular exponentiation where no special-case optimizations are applicable. This is only required for the calculation of t'_1 and t'_2 when producing a BDLEQ proof (see Section 8.2.2.3), or t_1 and t_2 when verifying the proof. It is also used by several higher-level functions in Safehouse. Due to the careful design of BRAKEM$^{\text{DDL}}_{\star}$, all other modular exponentiations can be optimized with one of the special cases.

4. Section 8.3.2.4 discusses an improved modular exponentiation technique when the exponent is fixed (i.e., available for precomputation). This is used whenever it is necessary to check group membership in $\mathbb{G}_2$, such as for each y_j and z_j in Π_0 (see Section 8.2.7). An element a is in $\mathbb{G}_2$ if $a^{p_3} = 1 \pmod{p_2}$, so precomputations can be performed to optimize for the fixed exponent p_3. Using this method also works to verify membership in $\mathbb{G}_1$, but it is far more expensive because p_2 is much larger than p_3. However, since $p_1 = 2p_2 + 1$, $\mathbb{G}_1$ is actually the set of quadratic residues in $\mathbb{Z}^*_{p_1}$. This means that group membership $\mathbb{G}_1$ can be checked almost instantly using the Legendre symbol: $v \in \mathbb{G}_1$ if and only if Legendre$(v/p_1) = 1$. Consequently, the "fixed exponent" special case is used only for $\mathbb{G}_2$ membership checks.

5. Section 8.3.2.5 discusses an improved modular exponentiation technique when the base is fixed. This is by far the most widely used optimization in $\text{BRAKEM}_x^{\text{DDL}}$ for fixed bases g_2, g_1, and x (the burner key). Exponentiation of g_1 is used by both Π_0 and $\text{BRAKEM}_\star^{\text{DDL}}$.Encapsulate. Exponentiation of g_2 is used commonly throughout the whole system, beginning with the $\text{BRAKEM}_\star^{\text{DDL}}$.KeyGen function. Exponentiation of x is used by the Π_0 prover to compute t' and the verifier to compute t, and by $\text{BRAKEM}_x^{\text{DDL}}$.Encapsulate to compute h_i (see Section 8.2.8). In total, a call to $\text{BRAKEM}_x^{\text{DDL}}$.Encapsulate requires $m \cdot (\ell + 1)$ exponentiations of x, which is a very large amount. In large Safehouse groups, typical values are $m = 10$ and $\ell = 128$, resulting in 1290 exponentiations of x for a single encapsulation. This underscores the importance of the burner key, which enables the "fixed base" special case optimizations.

6. Section 8.3.2.6 discusses an improved modular exponentiation technique for when the result of several exponentiations must be multiplied together. This technique is used to verify a Π_0 proof when computing t and T. It is also used to compute t_2' when producing a BDLEQ proof, and t_2 when verifying it.

7. Section 8.3.2.7 discusses opportunities for parallelism throughout the scheme. There are many parts of the system that can take advantage of modern multi-core processors.

All benchmarks presented in this section were performed 100 000 times using all Skylake cores on an Intel Core i7-6700K CPU (four physical cores with hyper-threading enabled), with all core frequencies pinned to 4.0 GHz and Intel Turbo Boost disabled.

8.3.2.1 Fast Modular Multiplication

Speeding up modular multiplication is greatly beneficial because it is the primitive operation at the core of modular exponentiation methods. The most straightforward approach to computing $a \cdot b \pmod p$ is to compute the product ab and then divide by p, taking the remainder. This approach is relatively slow because the division step is expensive. It is possible to improve the performance by considering the intended use: modular exponentiation requires a long sequence of multiplications. Consequently, it is beneficial to spend some extra time doing pre- or post-computation if it enables a faster multiplication step.

Montgomery multiplication [Mon85] is a well-known technique to accelerate modular multiplication within an exponentiation algorithm. The technique takes advantage of the fact that it is very efficient to divide numbers by powers of two: $a/2^s$ is equivalent to a bit-shifted right by s bits. Given a parameter $R = 2^s$ such that $\gcd(R, p) = 1$ (which is always true for a prime modulus $p > 2$) and an input a, the *Montgomery reduction* algorithm (also known as REDC) produces

$\texttt{REDC}_p(a) = aR^{-1} \pmod{p}$. REDC is very efficient because it only performs operations modulo R instead of modulo p, and $R \ll p$ for cryptographic applications. Additionally, since R is a power of two, computing a value modulo R merely involves discarding the uppermost bits. $\texttt{REDC}_p$ makes use of the value $-p^{-1} \pmod{R}$, which can be precomputed for a given p.

To implement multiplication modulo p, the algorithm can compute ab and return $\texttt{REDC}_p(ab)$. Note that this procedure returns $abR^{-1} \pmod{p}$ instead of the desired $ab \pmod{p}$, which may initially seem unhelpful. This challenge can be overcome by converting the inputs into *Montgomery form* by multiplying them by R beforehand:

$$\text{MM}_p(aR, bR) := \texttt{REDC}_p((aR) \cdot (bR)) = (aRbR) \cdot R^{-1} \pmod{p} = abR \pmod{p}$$

The Montgomery multiplication function MM_p is *stable*: given inputs in Montgomery form, it returns the product modulo p in Montgomery form. Converting a regular integer a to Montgomery form modulo p can be done by computing $\text{MM}_p(a, R^2)$. Converting a in Montgomery form back into a regular integer can be done by computing $\text{MM}_p(a, 1)$. Since exponentiation involves a long sequence of modular multiplications, the inputs can be converted to Montgomery form at the start of the sequence and then converted back at the end of the sequence.

In practice, the cryptographically large numbers being multiplied are too large to fit into a single machine word (typically 64 bits). The Montgomery multiplication process can be extended to support arbitrary precision integers—values stored as a sequence of machine words called *limbs*. When using Montgomery multiplication, the limbs are typically *saturated*—all bits in each limb are used to store the value.[20] An additional performance improvement can be gained by using "almost" Montgomery multiplication [Gue12]. In the standard case, numbers in Montgomery form are guaranteed to be less than p due to a conditional subtraction at the end of $\texttt{REDC}_p$. Numbers in "almost Montgomery form" are instead guaranteed to be less than 2^n for a parameter n. The only difference is that the condition for the subtraction in $\texttt{REDC}_p$ is faster to check with this optimization. Values in "almost Montgomery form" remain stable.

Finally, it is also possible to optimize the initial computation of the product $(aR) \cdot (bR)$ on the Intel x86-64 architecture by taking advantage of the ADX and BMI2 instruction set extensions [OGGF12]. ADX provides the ability to perform two add-with-carry chains simultaneously with the ADCX and ADOX instructions, and the MULX instruction provided by BMI2 enables multiplication without affecting the source operands or the flags. Together, these can be used to implement an extremely efficient arbitrary precision "multiply and add" operation. The CPUID

[20] It may be possible to further accelerate the performance of $\text{BRAKEM}_\star^{\text{DDL}}$ by using *unsaturated* limbs with partial modular reductions if the primes p_2, p_1, and p_0 are chosen to have a special structure. Erbsen et al. [EPG+19] describe several such approaches; exploring these is left to future work.

feature of the Intel architecture can be used to probe for ADX and BMI2 support (which is now commonly available) and fall back to standard `MUL` and `ADC` chains with flag storage if necessary.

The Go standard library includes support for arbitrary precision integers. It implements modular exponentiation using almost Montgomery multiplication [Gue12, Fig. 3] and ADX/BMI2-enabled assembly. Rather than using these functions directly, the prototype implementation of BRAKEM$^{\text{DDL}}_{\star}$ uses a modified copy of the code. This enables two important optimizations: caching the precomputed Montgomery parameters (including $-p^{-1} \pmod R$) between exponentiation calls, and keeping values in Montgomery form for longer. The latter optimization is beneficial for the more advanced exponentiation special cases.

8.3.2.2 Fast Modular Squaring

When working with very large exponents, particularly numbers with low Hamming weights like the sample primes in Section 8.3.1.4, the majority of execution time during an exponentiation is spent performing modular squaring. Luckily, modular squaring can be performed more quickly than modular multiplication by taking advantage of redundancy in the operands.

When computing $A \times B$ with arbitrary precision integers, each limb in A must be multiplied with each limb in B. These intermediate double-width products are then added to the result in the correct bit position: the double-width product $A_i \times B_j$ is added to limbs S_{i+j} and S_{i+j+1} (for the low and high word, respectively) in the result S. In the squaring case, $A = B$, so both $A_i \times A_j$ and $A_j \times A_i$ are added to S_{i+j} (for both low words) and S_{i+j+1} (for both high words). Since multiplication is commutative, this means that time can be saved in every case where $i \neq j$ by computing the intermediate product once and adding it to the result twice; this is far more efficient than performing two multiplications.

Ozturk et al. [OGG13] outlined a fast squaring procedure for x86-64 using the ADX and BMI2 extensions. However, their abstract presentation is not immediately translatable to an algorithm. Figure 8.2 depicts a modified version of their diagram for squaring a 512-bit 8-limb operand A, producing $S = A^2$, in an arrangement that makes the loop indices clear.

The BRAKEM$^{\text{DDL}}_{\star}$ prototype implements this fast squaring procedure using hand-written assembly.[21] To implement modular squaring, an operand in (almost) Montgomery form is squared using the fast squaring procedure and then reduced using $\texttt{REDC}_p$. The intermediate products ${A_i}^2$ are computed using an ordinary (non-ADX) `MUL` instruction and moved into place in the buffer

[21] Go provides an inline pseudo-assembly language that is slightly more portable than native x86-64 assembly. It provides light abstractions that help to integrate with the rest of language, while also allowing the entry of raw machine code if desired. See `https://golang.org/doc/asm` for details.

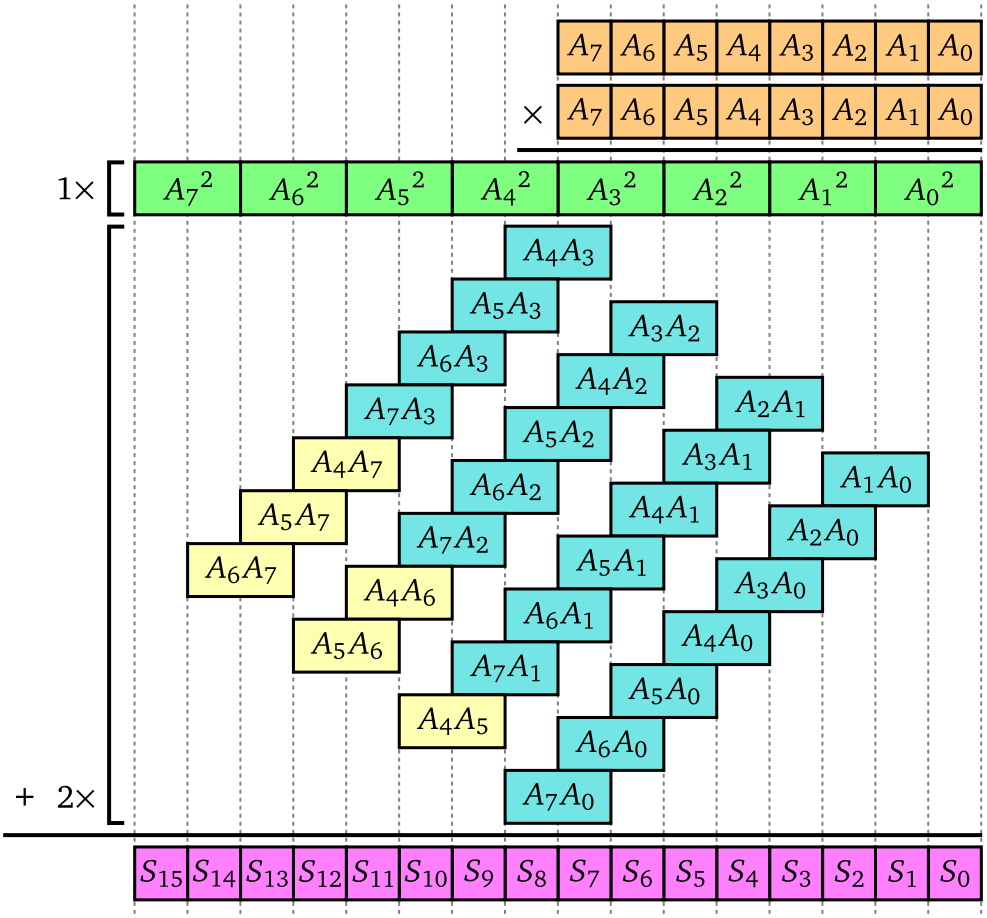


Figure 8.2 ARBITRARY PRECISION FAST SQUARING PROCEDURE. This figure is adapted from Intel's white paper [OGG13, Fig. 4]. It depicts a 512-bit input value A being squared to produce a 1024-bit output value $S = A^2$. Vertical rules denote word boundaries, and vertical alignment indicates which words are added together. The two different shades for the double-width products on the diagonals indicate the two different mechanisms for computing operand indices. (Refs: 225 and 227[a b])

for S. Each diagonal in Figure 8.2 is computed using a `MULX`/`ADCX`/`ADOX` chain. This operation is similar to the one in Section 8.3.2.1 (and described by Ozturk et al. [OGGF12]), except that the operands are changed partway through the diagonal. For an input integer with m limbs, diagonal $i \in [0, m/2)$ begins by multiplying A_i with $A_{i+1}, \ldots, A_{m-1}$, at which point it resumes the chain by multiplying $A_{(m/2)+i}$ with $A_{m/2}, \ldots, A_{(m/2)+i-1}$. The products from each of these diagonals are added to a buffer S'. Finally, S' is added to S twice. This can also be sped up using the `ADCX` and `ADOX` instructions: adding S'_i to S_i with each instruction loads the appropriate carry bit into both `CF` and `OF`, automatically handling adding the carry bit twice. The final Montgomery reduction step can also be simplified: since A^2 is already stored in S, the result can be multiplied by $-p^{-1} \pmod{R}$ without having to multiply the input values simultaneously. However, the carry bit of this multiplication must still be propagated through S correctly; this is done most efficiently using a simple `ADC` and `INC` loop.

There is an edge case that occurs in this squaring procedure when the number of limbs in the operand is odd. When this occurs, the first (i.e., bottom-rightmost) diagonal depicted in Figure 8.2 contains an extra product, and the chains do not switch operands until diagonal 2 (instead of diagonal 1).

8.3.2.3 Exponentiation: Variable Base and Exponent

As discussed in Section 8.3.2, BDLEQ requires an implementation of modular exponentiation with a variable base and variable exponent. Since both the base and exponent change for each exponentiation, no precomputation involving these values is possible; only the modulus is available to precomputation techniques. For this application, the exponent is drawn uniformly at random from $[2, p_3)$.

A simple way to implement modular exponentiation is to use Algorithm 8.3 with Montgomery multiplication (see Section 8.3.2.1) and squaring (see Section 8.3.2.2). Montgomery parameters, including R, are precomputed for the (fixed) modulus p. When asked to compute an exponentiation $z = v^y \pmod{p}$, the variable base v is first converted into Montgomery form using $v \leftarrow \mathrm{MM}_p(v, R^2)$. The algorithm then proceeds as normal, setting $z \leftarrow R$ initially, then using $\mathrm{MM}_p(z, v)$ to perform multiplication and $\mathrm{MM}_p(z, z)$ to perform squaring (with the optimized square computation described in Section 8.3.2.2). Finally, the result is converted to regular form by returning $\mathrm{MM}_p(z, 1)$.

This approach works, but it is not the only way to implement exponentiation. In fact, it is typically one of the least efficient options for the BDLEQ use case. Good summaries of the available options are given in the Handbook of Applied Cryptography [MOV97, §14.6.1], The Art of Computer Programming [Knu97, §4.6.3], and by Gordon [Gor98]. Algorithm 8.3 is often

$$50971 = 1100011100011011$$
$$12 \quad 7 \quad 1 \quad 11$$
$$\left(\left(12 \cdot 2^4 + 7\right) \cdot 2^4 + 1\right) \cdot 2^4 + 11$$
$$v^{50971} = \left(\left(\left(v^{12}\right)^{2^4} \cdot v^7\right)^{2^4} \cdot v^1\right)^{2^4} \cdot v^{11}$$

Figure 8.3 EXAMPLE OF FIXED-WINDOW EXPONENTIATION. This is the computation of v^{50971} using a window size of $k = 4$. The exponent is written in binary and partitioned into k-bit chunks. Each window y_i defines the partial product v^{y_i}. These products are multiplied into the accumulator from most- to least-significant bit. The accumulator is squared k times between multiplications. (Refs: 228, 230, and 231)

called the "binary method", since it interprets the exponent in binary form and determines when to multiply the accumulator by scanning the individual bits. A natural generalization is to write the exponent y in a different base m:

$$y = \sum_{i=0}^{\lfloor \log_m(y) \rfloor} y_i m^i$$

Given this representation of y with terms $y_i \in [0, m)$, the "m-ary method" initially computes the powers $v^2, \ldots, v^{m-1}$, replaces the squaring in Algorithm 8.3 with exponentiation by m, and replaces the multiplication when $b = 1$ with multiplication by v^{y_i}. Consequently, the m-ary method processes the exponent in "chunks" of size m, rather than one bit at a time. The powers of v that are computed initially are called the *dictionary*.

When $m = 2^k$, the implementation of the m-ary method becomes more convenient, since computing z^m is equivalent to squaring the accumulator k times, and computing the components y_i can be done with efficient bitwise operations. The resulting algorithm can be viewed as breaking the exponent y into "windows", each containing k bits. For this reason, the 2^k-ary method, which was first proposed by Brauer [Bra39], is often called the "fixed window" method. Figure 8.3 depicts an exponentiation using the fixed-window method. When combined with Montgomery multiplication, the fixed window method is efficient enough for a practical BDLEQ implementation.

The final improvement that is discussed in this section is a natural extension of the fixed window method based on the observation that the windows do not need to be adjacent to each other. The "sliding window" method [Thu73] places variable-width windows freely within the binary representation of the exponent. Figure 8.4 depicts an exponentiation using the

Algorithm 8.4 SLIDING WINDOW MODULAR EXPONENTIATION. The dictionary dict is a map from integers to group elements. dict$[n]$ denotes the entry in the dictionary indexed by integer n. $y[i]$ denotes the i^{th} bit of the binary representation of y, where the least significant bit has index 0. $y[i..j]$ denotes the bits of y indexed between i and j (inclusive) extracted as an integer (e.g., if $y = 12$, then $y[3..2] = 3$). (Ref: 230)

Subroutine | SlidingWindowModExp$_k(v, y, p) \rightarrow z$

Output: $z = v^y \pmod p$.

dict$[1] \leftarrow v$. ▹ Form the dictionary.
dict$[2] \leftarrow v \cdot v \pmod p$.
for $(i \leftarrow 3; i < 2^k; i \leftarrow i + 2)$ {
  dict$[i] \leftarrow$ dict$[i-2] \cdot$ dict$[2] \pmod p$.
}
$z \leftarrow 1$. ▹ Begin processing windows.
$i \leftarrow \lceil \log_2(y) \rceil - 1$.
while $(i \geq 0)$ {
  while $(y[i] =$ "0") { ▹ Skip zeroes between windows.
    $z \leftarrow z \cdot z \pmod p$.
    $i \leftarrow i - 1$.
    if $(i < 0)$ **return** z.
  }
  $\ell \leftarrow k - 1$. ▹ Compute window length ℓ.
  while $(i - \ell < 0 \,||\, y[i - \ell] =$ "0") {
    $\ell \leftarrow \ell - 1$.
  }
  repeat $(\ell + 1$ times) { ▹ Make space for the window.
    $z \leftarrow z \cdot z \pmod p$.
  }
  $z \leftarrow z \cdot$ dict$[y[i..i-\ell]] \pmod p$. ▹ Multiply the window.
  $i \leftarrow i - \ell - 1$.
}
return z.

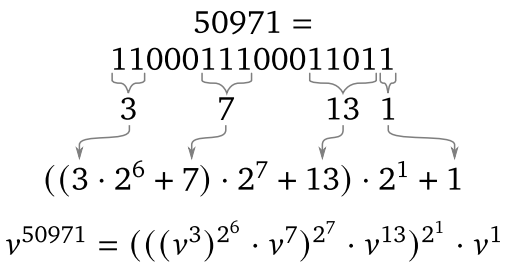


Figure 8.4 EXAMPLE OF SLIDING-WINDOW EXPONENTIATION. This is the computation of v^{50971} with a maximum window size of $k = 4$, constructed by scanning from most- to least-significant bit. The number of times that the accumulator is squared after a multiplication varies with the window placement: one squaring is performed for each "0" bit between the windows, and one squaring is performed for each bit contained in the subsequent window. Figure 8.3 depicts the fixed-window version of the same example. (Refs: 228 and 232)

sliding-window method. While the windows no longer need to be adjacent, they still need to be non-overlapping. Each window is at least one bit long and at most k bits long. Every window always ends with a "1" bit, since trailing "0" bits can be omitted by shortening the window. This means that unlike the fixed window method, which must initially compute all powers of v less than 2^k for its dictionary, the sliding window method only needs to construct odd powers of v less than 2^k: $v, v^3, \ldots, v^{2^k-1}$. A high-level overview of the sliding window algorithm is given in Algorithm 8.4.

The prototype BRAKEM$^{\text{DDL}}_{\star}$ implementation uses the sliding window method with $k = 4$. This was empirically found to be the most efficient approach for the BDLEQ exponentiations with the given group parameters. For the 253-bit p_3 defined in Section 8.3.1.4, the method uses at most 64 windows for an exponent in $[2, p_3)$. Since the dictionary contains eight odd powers of v and each window contains an odd 4-bit integer selected uniformly at random, every entry of the dictionary will usually be accessed—this means that there is no practical advantage to pruning the dictionary based on the actual window contents. More sophisticated exponentiation algorithms are available that theoretically require fewer multiplications, but they typically require more expensive computation when deciding what values to multiply; this additional expense undermines their theoretical performance benefits in this setting.

8.3.2.4 Exponentiation: Fixed Exponent

All of the methods for exponentiation discussed in Section 8.3.2.3 are ultimately just methods for determining what multiplications to perform. Consider one of the fundamental identities of

exponentiation, which also applies to modular arithmetic:

$$v^{y_1} \cdot v^{y_2} = v^{y_1+y_2}$$

All of the exponentiation algorithms involve multiplications with a single base, v. This means that an alternative view is that they are algorithms for determining which *exponents* to *add*. When two powers of v are multiplied as part of the sequence, they yield a new power of v whose exponent is the sum of the two original exponents. This means that the sequence of multiplications dictated by the exponentiation algorithm corresponds to a sequence of exponents, where each value is the sum of two values that appeared previously in the sequence. This is exactly the definition of an *addition chain*. Consequently, minimizing the number of multiplications required to compute v^y (and therefore computing the result as quickly as possible) is equivalent to the problem of finding the minimum addition chain for y. Since squaring is more efficient than general multiplication in this setting (see Section 8.3.2.2), the minimization problem can be further adjusted to treat squaring operations (corresponding to adding a previous value to itself in the addition chain) as less expensive.

To see the relationship between exponentiation and addition chains, consider the running example $v^{50971} \pmod{p}$ from Section 8.3.2.3. Note that the binary representation of 50971 is 1100011100011011. The binary method (see Algorithm 8.3) computes the result as follows:

$$v^{50971} = ((((((v^2 \cdot v)^{2^4} \cdot v)^2 \cdot v)^2 \cdot v)^{2^4} \cdot v)^2 \cdot v)^{2^2} \cdot v)^2 \cdot v$$

Examining the exponents as they accumulate reveals that this multiplication sequence is associated with the following addition chain:

$$(1, 2, 3, 6, 12, 24, 48, 49, 98, 99, 198, 199, 398, 796, 1592, 3184, 3185, 6370, 6371, 12742, 25484, 25485, 50970, 50971)$$

Note that each value in the chain is the sum of two previous values.[22] In contrast, the fixed window method depicted in Figure 8.3 produces this addition chain:

$$(\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, \underline{13}, \underline{14}, \underline{15}, 24, 48, 96, 192, 199, 398, 796, 1592, 3184, 3185, 6370, 12740, 25480, 50960, 50971)$$

[22] The binary method produces addition chains where each value is either the previous value added to itself, or the previous value added to 1.

The underlined values indicate values that are involved in the construction of the dictionary. The sliding window method depicted in Figure 8.4 produces this addition chain:

$$(\underline{1}, \underline{2}, \underline{3}, \underline{5}, 6, \underline{7}, \underline{9}, \underline{11}, 12, \underline{13}, \underline{15}, 24, 48, 96, 192, 199, 398, 796, 1592, 3184, 6368, 12736, 25472, 25485, 50970, 50971)$$

Note that for the sliding window method, the dictionary contains only odd values (and 2, which is used to generate the dictionary).

While these addition chains are not minimal, the algorithms that generate the chains are extremely efficient. This is important for "ordinary" exponentiation where chain generation must be done as part of the exponentiation algorithm, but it is not important when the exponent is known in advance. In this case, it is possible to spend a long time searching for a short addition chain. This chain can then be cached. As discussed in Section 8.3.2, exponentiation by a fixed exponent is used in $\text{BRAKEM}_{\star}^{\text{DDL}}$ to test for membership in $\mathbb{G}_2$. If an integer in $\mathbb{Z}_{p_2}^*$ equals 1 (mod p_2) after exponentiation by p_3, then it is a member of $\mathbb{G}_2$. Consequently, this membership test can be accelerated by finding an addition chain for p_3 that is shorter than the one produced by the sliding window algorithm with $k = 4$.[23]

Short addition chains for use in fixed-exponent exponentiation are traditionally found by hand, which is a long and tedious process that may not produce good results. The Handbook of Applied Cryptography [MOV97, §14.6.2] and the Handbook of Elliptic and Hyperelliptic Curve Cryptography [CFA+05, §9.2] both provide rudimentary summaries of automated techniques for finding these chains. The publicly available `addchain` software [McL20] incorporates a wide variety of published techniques to find short addition chains, and it often matches or surpasses hand-generated chains. Several open-source contributions to the `addchain` project were made as part of this work.

`addchain` was able to find a variety of short addition chains for p_3. The efficiency of addition chains for exponentiation can be measured in terms of the number of doublings and non-doublings they perform, corresponding to squarings and multiplications in the exponentiation algorithm, respectively. Empirical measurements of the prototype implementation showed that in $\mathbb{G}_2$, the fast squaring method described in Section 8.3.2.2 completes in approximately 90% of the time required for a multiplication. Using this weighting, the best addition chain found by `addchain` requires 251 squarings and 31 multiplications. This chain was found using the sliding window method with $k = 8$ to identify terms to include in the dictionary [CFA+05, §9.2.2]. The chain was then built from the dictionary using the "continued fractions" method [BBBD89] with the "binary" strategy [BBB94].

[23] ^ This is the chosen algorithm for "ordinary" exponentiation, as discussed in Section 8.3.2.3.

The output of `addchain` is a program in a domain-specific language describing the squarings and multiplications to perform. The language is very simple: it assigns values to variables based on additions, doublings, and bit shifts (representing multiple doublings) of previous variables or 1, and finally returns the target value. This program can be parsed by the `addchain` library. The addition chain program for p_3 is as follows:

```
      _10=2*1
      _11=1+_10
     _100=2*_10
     _110=2*_11
    _1000=2*_100
    _1011=_11+_1000
   _10000=2*_1000
  _100000=2*_10000
  _100110=_110+_100000
 _1000000=2*_100000
 _1010000=_10000+_1000000
 _1010011=_11+_1010000
 _1100011=_10000+_1010011
 _1100111=_100+_1100011
 _1101011=_100+_1100111
_10010011=_1000000+_1010011
_10010111=_100+_10010011
_10111101=_100110+_10010111
_11010011=_1000000+_10010011
_11100111=_1010000+_10010111
_11101101=_110+_11100111
_11110101=_1000+_11101101
     i160=((_1011+_11110101)<<126+_1010011)<<9+_10
     i179=((_11110101+i160)<<7+_1100111)<<9+_11110101
     i209=((i179<<11+_10111101)<<8+_11100111)<<9
     i232=((_1101011+i209)<<6+_1011)<<14+_10010011
     i263=((i232<<10+_1100011)<<9+_10010111)<<10
   return ((_11110101+i263)<<8+_11010011)<<8+_11101101
```

This addition chain program corresponds to an exponentiation algorithm. As part of the prototype implementation, the program was converted into native Go code that performs modular multiplication and squaring as needed. Table 8.1 presents a comparison between three methods for exponentiating by p_3: the Go standard library's modular exponentiation function, the optimized variant in Section 8.3.2.3, and the fixed exponent method using the aforementioned addition chain. The optimized addition chain provides a small but measurable performance improvement.

Table 8.1 A PERFORMANCE COMPARISON OF FIXED-EXPONENT EXPONENTIATION ALGORITHMS. The fixed exponent used in the comparison is p_3. Standard deviations are in parentheses. (Ref: 233)

Algorithm	Time [ms]
Go standard library	0.95 (0.01)
"Ordinary" (Section 8.3.2.3)	0.674 (0.009)
Fixed exponent (Section 8.3.2.4)	0.661 (0.006)

8.3.2.5 Exponentiation: Fixed Base

Of all of the operations performed by $\text{BRAKEM}_{\star}^{\text{DDL}}$, it is most important to optimize modular exponentiation of a fixed base (i.e., a base for which precomputations are applicable). Precomputation should be performed to accelerate exponentiation of three fixed bases: g_2, g_1, and the burner key x for $\text{BRAKEM}_{x}^{\text{DDL}}$. Accelerating exponentiation of g_1 is particularly important, since exponents for $\mathbb{G}_1$ are 3072 bits long.

Section 8.3.2.4 described how exponentiation algorithms correspond to addition chains. A fixed-base exponentiation algorithm operates by precomputing powers of the fixed base, and then reusing these cached results to perform exponentiations. This process can be viewed in terms of addition chains: the precomputed powers are formed using an addition chain, and then a subset of this chain can be included in subsequent chains for free (i.e., without inducing any multiplications). In theory, the optimal exponentiation algorithm corresponds to the minimum-length addition chain that contains all of the outputs that are ever requested. Fixed-base exponentiation is therefore closely associated with the *multi-exponentiation* problem: computing $(g^{y_1}, \ldots, g^{y_\ell})$ more efficiently than performing ℓ independent exponentiations.

The academic history of multi-exponentiation algorithms is complex; attribution for many of the ideas is controversial [Ber02]. Yao [Yao76] was among the first to describe an exponentiation algorithm with a much more efficient addition chain than the "binary method" (see Algorithm 8.3). Yao's algorithm is related to the "m-ary method" discussed in Section 8.3.2.3 through an algorithm transposition process [Ber02, §5]. The algorithm was later presented with minor improvements (eliminating unnecessary elements at the start of the chain and an improved summation method for elements at the end of the chain) as a textbook exercise answer by Knuth [Knu97, answer to §4.6.3 exercise 9]. Contemporaneously with Yao's publication, Pippenger [Pip76] presented a constructive proof of a lower bound for the multi-exponentiation problem, followed by several improvements [Pip78; Pip80]. Pippenger's algorithm[24] is nearly optimal, but it remains largely overlooked—subsequent reinventions of various aspects of the technique are cited far more

[24] ^ Pippenger's algorithm refers to the algorithm implicitly associated with the constructive proof. Interested readers should refer to Henry [Hen10] for the most accessible definition of the algorithm.

frequently. Yao's algorithm is often misattributed to Brickell et al. [BGMW93], hereafter referred to as BGMW. However, unlike the previous presentations of the algorithm, BGMW explicitly noted that the algorithm was amenable to precomputation—the start of the addition chain is independent of the desired exponent. Moreover, the BGMW algorithm is a generalization of Yao's algorithm: it includes an additional parameter called M, a set of multipliers. The variant of Yao's algorithm described by Knuth is exactly equal to the BGMW algorithm when $M = \{1\}$. A patent was granted in the USA for the BGMW algorithm; this patent expired in 2012.

The aforementioned algorithms all include parameters that control a time-vs-space tradeoff: precomputing more powers of the fixed base uses more storage space, but can help to shorten the per-exponent fragments of the addition chain. The 1993 BGMW extended abstract included a table of suggested parameters [BGMW93, Tbl. 2] that control this tradeoff. In an unpublished 1995 version of the paper, Brickell et al. included additional rows in the table [BGMW95, Tbl. 2]. The new final row contains a set of parameters that fundamentally alters the algorithm in order to achieve the fastest exponentiation with the most storage space. While this configuration was likely included as a theoretical exercise to illustrate extreme behavior,[25] it also happens to represent the best parameter shape for modern machines implementing BRAKEM$^{\text{DDL}}_{\star}$. When Brickel et al. first suggested these parameters, typical personal computers owned by consumers contained between 1 and 128 MB of RAM. In 2021, a typical smartphone or PC is likely to contain 8 or 16 GB of RAM, with operating system services, web browsers, and browser-based messaging applications allocating several gigabytes of virtual memory in order to provide basic functionality. Consequently, it is acceptable and worthwhile for modern secure messaging applications to allocate hundreds of megabytes for precomputed group elements if doing so accelerates messaging interfaces and lowers battery use by computing exponentiations more quickly.

The algorithm used by the BRAKEM$^{\text{DDL}}_{\star}$ prototype to implement fixed-base exponentiation is much easier to describe than it is to attribute. The algorithm is parameterized by a block size k. The exponentiation algorithm is simply the 2^k-ary method (also known as the "fixed window" method) described in Section 8.3.2.3. The number of blocks is $\ell = \lceil \lceil \log_2(p) \rceil / k \rceil$, where every exponent y is in $[0, p)$. For a given base g, the algorithm precomputes all of the following values:

$$\{(g^y)^{2^{i \cdot k}} \mid 0 \leq i < \ell \wedge 1 \leq y < 2^k\}$$

When asked to compute $z = g^y$, y is first written as:

$$y = \sum_{i=0}^{\ell-1} \left(y_i \cdot 2^{i \cdot k} \right)$$

[25] ^ The text in the paper does not reference these parameters at all.

Table 8.2 A PERFORMANCE COMPARISON OF FIXED-BASE EXPONENTIATION ALGORITHMS. The algorithms were used to compute $z = g^y \pmod m$ where $y \in [0, p)$. (Refs: 236 and 239)

g	p	m	k	ℓ	$\lvert g\rvert$ [B]	Mem [MiB]	Time [ms]	
g_1	p_2	p_1	11	280	392	214.3	0.21	(0.02)
g_2	p_3	p_2	11	23	384	17.3	0.050	(0.007)
Burner key (client)	p_3	p_2	11	23	384	17.3	0.050	(0.007)
Burner key (server)	p_3	p_2	6	43	384	1.0	0.07	(0.01)
Sliding window, g_1	p_2	p_1	-	-	-	0	8.82	(0.06)
Sliding window, g_2	p_3	p_2	-	-	-	0	0.72	(0.01)

Standard deviations are in parentheses. $\lvert g\rvert$ denotes the size of group elements in memory. "Mem" denotes the total storage space for all precomputed values. The "sliding window" rows represent the sliding window algorithm with a maximum window size of 4, as described in Section 8.3.2.3, with no precomputation.

This is equivalent to partitioning the binary representation of y into ℓ k-bit blocks with the most significant window shortened if k does not divide $\lceil \log_2(p) \rceil$. The result can then be written as:

$$z = g^y = \prod_{i=0}^{\ell-1} (g^{y_i})^{2^{i \cdot k}}$$

When asked to compute $z = g^y$, the appropriate cached values are simply multiplied together. Every exponentiation can be computed with at most $\ell - 1$ multiplications using this approach (any $y_i = 0$ can be ignored). This algorithm is equivalent to the one described by Brickell et al. [BGMW95, §3] with $b = 2^k$, $M = \{1, 2, 3, \ldots, 2^k - 1\}$ and $h = 1$.[26]

The prototype implementation of BRAKEM$^{\text{DDL}}_{\star}$ uses different values of k for the different fixed bases. Since g_2 and g_1 are constant for all group conversations in a deployment, a large amount of precomputation can be performed. Both the client and server set $k = 11$ for these bases. Since the burner key for BRAKEM$^{\text{DDL}}_{x}$ varies between conversations when following the advice in Section 8.3.1.3, a client will likely need to exponentiate a few fixed burner keys at any given time, while a server will need to exponentiate far more (one for each group conversation it manages). Consequently, clients use $k = 11$ for burner key precomputations, while servers reduce this to $k = 6$. Table 8.2 shows the relevant sizes and memory usage for the precomputation tables. With these settings, a client uses $(231.6 + 17.3n)$ MiB of memory when connected to n groups, and a server uses $(231.6 + n)$ MiB of memory to manage n groups. In exchange for this memory use, exponentiations of g_2 and x are an order of magnitude faster, and exponentiations of g_1 are

[26] Yao's algorithm [Yao76] corresponds to the "basic strategy" described by Brickell et al. [BGMW95, §2]. In the notation of the full BGMW algorithm, this corresponds to $b = 2^k$, $M = \{1\}$, and $h = 2^k - 1$.

approximately 4 times faster. For a deployment scenario with different memory requirements, the value of k can be adjusted, or a different fixed-base exponentiation algorithm can be used.

Each call to $\text{BRAKEM}_x^{\text{DDL}}$.Encapsulate performs multiple independent fixed-base exponentiations: g_2, g_1, and x are each exponentiated once to produce Π_0, and once for each ring of receivers. Pippenger's algorithm [Pip76] could theoretically improve this performance even further. Unfortunately, Pippenger's algorithm only performs well when configured with appropriate parameters, and it is not easy to find these parameters for realistic problems [Hen10, §4]. Consequently, Pippenger's algorithm is not yet practical for real-world applications.

8.3.2.6 Product of Exponentiations

As discussed in Section 8.3.2, several algorithms in $\text{BRAKEM}_\star^{\text{DDL}}$ involve computing a product of exponentiations.[27] Given group elements $(v_1, \ldots, v_n)$ and exponents $(y_1, \ldots, y_n)$, the following expression must be computed:

$$\prod_{i=1}^{n} {v_i}^{y_i}$$

The obvious way to compute this quantity is to use the "ordinary" exponentiation discussed in Section 8.3.2.3 to evaluate each ${v_i}^{y_i}$ term, and then to multiply the results. This is a reasonable approach, but it turns out that there is a better method. Straus [Str64] was the first to note that Brauer's 2^k-ary exponentiation method [Bra39] could be generalized to compute a product of exponentiations more efficiently than computing the terms separately. ElGamal [ElG85, §V.B] popularized the technique when he described a special case of Straus' algorithm for the purpose of speeding up his signature scheme.

Straus' algorithm takes advantage of the fact that the naïve approach performs up to $\lceil \log_2(p) \rceil$ squaring operations for each term, where exponents are drawn from $[0, p)$, only for the final results to be multiplied at the end. Time can be saved by "reusing" squarings for all of the terms. As an example this approach, consider the problem of computing ${v_1}^5 \cdot {v_2}^6$. Sliding window exponentiation with $k = 1$ could be used to compute each term, followed by a final multiplication:

$$\left({v_1}^{2^2} \cdot v_1\right) \cdot \left({v_2}^2 \cdot v_2\right)^2$$

This approach requires 4 squarings and 3 multiplications. Consider this alternative:

$$\left(\left(v_1 \cdot v_2\right)^2 \cdot v_2\right)^2 \cdot v_1$$

[27] This problem is sometimes referred to as "multi-exponentiation", but this term also refers to the general problem of efficiently performing multiple independent exponentiations (i.e., without restricting attention to their product).

The alternative approach requires only 2 squarings and 3 multiplications. Note that the savings come from multiple intermediate products, including both ${v_1}^{2^2}$ and ${v_2}^{2^2}$, being incorporated into the final result "simultaneously".

Straus' algorithm is parameterized by a window size k. It begins by computing a dictionary containing every possible product of the bases with exponents less than 2^k:

$$\left\{ \prod_{i=1}^{n} {v_i}^{e_i} \;\middle|\; \forall_{1 \le i \le n}\ 0 \le e_i < 2^k \right\}$$

The algorithm then proceeds similarly to the 2^k-ary method: the exponents are broken into k-bit windows and processed from the most significant bits to the least significant bits. During each iteration, the contents of the windows for each of the n exponents are examined simultaneously. This yields a vector $\vec{y} = \{e_1, \ldots, e_n\}$, where e_i contains the k-bit piece of y_i within the window. $\vec{y}$ is then used to look up the corresponding product in the dictionary. This value is multiplied into the running result. The result is then squared k times to move to the next window.

When implementing $\text{BRAKEM}_\star^{\text{DDL}}$, it is necessary to compute the product of exponentiations in several cases:

- $n \le 10$ in practice, $v_i \in \mathbb{G}_2$, $y_i \in [2, p_3)$ (computing T and T' in Π_0);
- $n \le 10$ in practice, $v_i \in \mathbb{G}_1$, $y_i \in [2, p_2)$ (computing t and t' in Π_0); and
- $n = 128$, $v_i \in \mathbb{G}_2$, $y_i \in [2, 2^{128})$ (computing t_2 and t_2' in BDLEQ).

In all of these cases, the most efficient approach in practice is to use Straus' algorithm with $k = 1$. Larger values of k cause too much time to be spent filling the dictionary with terms that are never used. Even if the algorithm is modified to compute dictionary values "on-demand", too many intermediate values must still be computed to make the larger k worthwhile [Ber02, §3]. Like the 2^k-ary method, Straus' algorithm can also be transformed into a "sliding window" version by skipping the multiplication step when the window is zero for all exponents. This is also not useful for $\text{BRAKEM}_\star^{\text{DDL}}$: n is large enough that this event is very rare, so the benefits from the additional complexity are negligible.[28] Pippenger's algorithm [Pip76] can also theoretically solve this problem with fewer operations, but as mentioned in Section 8.3.2.5, it is not clear that the algorithm can perform well in practice.

[28] Although a 2^{-10} probability of skipping a multiplication in Π_0 with $k = 1$ is not negligible in cryptographic terms, the benefit is rare enough that careful profiling would be required to ensure that the extra conditional statement does not negate the benefits due to branch misprediction. This could be particularly costly if the same code was used to implement the $n = 128$ case for BDLEQ.

8.3.2.7 Parallelism

To achieve maximum performance on modern processors, algorithms must be parallelized. This section points out multiple opportunities for parallelism in BRAKEM$^{\text{DDL}}_{\star}$ implementations.

When designing a parallelized workflow, it is important to consider the performance characteristics of the chosen concurrency mechanisms. If there is a high cost associated with starting a parallel thread of execution, then the concurrency should be arranged so that each thread executes a larger work unit. Since the prototype implementation was written in Go, it is relatively inexpensive to launch concurrent tasks.[29] The prototype takes advantage of this concurrency model by using two strategies: parallelizing the exponentiation algorithms, and high-level parallelism. Implementations using traditional threading libraries may not see any benefit from parallelizing exponentiations, in which case more focus should be placed on high-level parallelism.

There is no worthwhile way to parallelize an individual exponentiation with a variable base, such as the "ordinary" exponentiation described in Section 8.3.2.3 or the fixed-exponent exponentiation described in Section 8.3.2.4. In both cases, almost all of the running time is spent evaluating an addition chain. The addition chain produced by the sliding window method is a *Brauer chain* (also known as a *star chain*): each term in the chain references the immediately preceding term. The fixed chain for p_3 is nearly a Brauer chain. Evaluating a Brauer chain is an inherently serial process, since each new term has a strict dependency on the previous term, so these methods cannot be meaningfully parallelized.

The fixed-base exponentiation method described in Section 8.3.2.5 is much easier to parallelize. Since $\ell \in \{23, 43, 280\}$ (see Table 8.2), the number of cached powers that must be multiplied usually exceeds the number of logical processor cores. Additionally, the order in which the values are multiplied does not matter, since there are no squaring operations to perform. Each core can be assigned to compute the product of a subset of terms. Once all of the work is complete, the final values computed by each thread of execution can then be "reduced" by multiplying them together. In the case of $\ell = 280$ on an 8-core machine, this would involve 8 parallel threads, each performing 34 multiplications, yielding 8 partial products, followed by 7 serial multiplications. These final multiplications can also be partially parallelized for an additional small performance boost if the overhead of the concurrency primitives is very small.

[29] The Go language includes support for concurrency in the form of *goroutines*, which are very lightweight threads of execution that are multiplexed onto actual OS-level threads. One of the reasons that goroutines are inexpensive to launch is that Go's memory model makes it safe to dynamically resize the stack, enabling goroutines to start with very small stacks.

Computing the product of exponentiations, as described in Section 8.3.2.6, can also be parallelized. It is trivial to split the problem into independent sub-problems. For example:

$$\prod_{i=1}^{n} {v_i}^{y_i} = \left(\prod_{i=1}^{j} {v_i}^{y_i}\right) \cdot \left(\prod_{i=j+1}^{n} {v_i}^{y_i}\right)$$

Each sub-problem can be can be solved using Straus' algorithm [Str64] in parallel, followed by a (potentially parallelized) reduction of the partial products to a single final result. Unfortunately, dividing the problem in this way undermines the benefit of Straus' algorithm, since squarings must be performed for each term. The extreme case of splitting the problem into n sub-problems effectively transforms the computation into the naïve algorithm. The best way to divide the problem into sub-problems depends on the value of n, the size of the exponents (which determines the number of squarings), and the performance characteristics of the processor. For the scenarios described in Section 8.3.2.6, empirical measurements using an Intel Core i7-6700K determined that it was most efficient to break the problem into two sub-problems when $n \leq 10$, and to break the problem into 8 (the number of logical cores) sub-problems when $n = 128$.

Finally, there are many opportunities for parallelization at higher layers of abstraction. These opportunities are easily found using a critical path analysis.[30] The following high-level computations can benefit from parallelism:

- In the BDLEQ prover (see Section 8.2.2.3), t'_1 and t'_2 (steps 4 and 5) can be computed in parallel.
- In the BDLEQ verifier, the two terms for t_1 (step 3) and the two terms for t_2 (step 4) can all be computed in parallel.
- In $\text{BRAKEM}^{\text{DDL}}_{\star}$.Encapsulate (see Section 8.2.8):
  - The calculations for each ring of receivers, including the BDLEQ proof, can be performed in parallel.
  - Within a ring:
    - The ElGamal ciphertexts can be computed in parallel.
    - k_i can be computed at the same time as the ciphertexts.
- In $\text{BRAKEM}^{\text{DDL}}_{\star}$.Verify (see Section 8.2.8), all of the NIZKPKs can be verified in parallel. Note that if the Fiat-Shamir challenges are merged into a single hash output to save space (as

[30] ^ For Go code, the `go tool trace` program can quickly find these bottlenecks.

suggested in Section 8.3.2), then the implementation should take care to ensure that the computed commitments are hashed in the correct order despite being computed in parallel.

- When producing a Π_0 proof (see Section 8.2.7):
  - The computation of t' (step 7) can be partially parallelized. The first $\ell \cdot m$ terms in the exponent of t' involve fixed-base exponentiations of the burner key x. These exponentiations can be computed in parallel, followed by a (potentially parallelized) summation. The other terms in the exponent (and for the exponent of T') involve only modular multiplications and are not worth parallelizing.
  - t' and T' (step 7) can be computed in parallel.
- When verifying a Π_0 proof:
  - t_0, T_0, E_1, E_2, and every t_j are all the sums of various terms computed in a loop with $\ell = 128$ iterations (step 5). These can be parallelized with the same technique used to compute the exponent for t' in the prover. Each thread of execution is responsible for processing a subset of the loop iterations and accumulating partial sums. The final sums can be computed with a (potentially parallelized) reduction.
  - t and T (steps 6 and 7) can be computed in parallel.
  - When computing t (step 6), ${g_1}^{t_0}$ can be computed at the same time as the product of the other terms.
  - When computing T (step 7), ${g_2}^{T_0}$ can be computed at the same time as the product of the other terms.

The prototype implementation of BRAKEM$^{\text{DDL}}_{\star}$ incorporates all of these techniques for parallelism. As a result, a typical BRAKEM$^{\text{DDL}}_{\star}$.Encapsulate or BRAKEM$^{\text{DDL}}_{\star}$.Verify operation is able to easily saturate the 8 available logical cores.

8.4 Security of BRAKEM$^{\text{DDL}}_{\star}$ (Σ)

This section discusses the security of the BRAKEM$^{\text{DDL}}_{\star}$ scheme presented in Section 8.2.8. Section 8.4.1 proves the security of the Π_0 NIZKPK described in Section 8.2.7, which is the core sub-protocol used by the BRAKEM construction. Section 8.4.2 sketches the security of the overall BRAKEM construction. Security proofs for the other DDL-based constructions discussed in Section 8.2.4—BRAKEM$^{\text{2DDL}}_{\star}$ and BRAKEM$^{\text{DDL}'}_{\star}$—can be easily formulated using the same approach.

Proving that Π_0 is secure is mostly straightforward, but the algebraic transformations require careful management of loop indices in order for the appropriate terms to cancel out. This section walks through the algebra in great detail for readers that would like to check the work. Proving that BRAKEM$^{\text{DDL}}_{\star}$ is secure requires the introduction and justification of new hardness assumptions, followed by a carefully constructed sequence of games. Readers that do not wish to verify the security proofs for the construction may wish to skip to Section 8.5.

8.4.1 Security of Π_0

8.4.1.1 Proof of Correctness

If both the prover and the verifier are honest and the statement is true, then the verifier will always accept the proof. Consider verifier step 2 in this scenario. For each $1 \le i \le \ell$: if $c'[i] = 0$ then u_i is computed exactly as u'_i in prover step 4 and v_i is set to $w_i = v'_i$ due to prover step 9.b; if $c'[i] = 1$ then v_i is computed exactly as v'_i in prover step 5 and u_i is set to $w_i = u'_i$ due to prover step 9.a. Therefore $u_i = u'_i$ and $v_i = v'_i$ in all cases. This means that the input to H_e in verifier step 3 is identical to the input used by the prover in prover step 6, and therefore both the prover and verifier use the same values for $e_1, \ldots, e_\ell, f_1, \ldots, f_\ell, F_1, \ldots, F_m$.

Let $\sum_i$ denote summation across $1 \le i \le \ell$ and $\sum_j$ denote summation across $1 \le j \le m$. Let $\sum$ denote the combined summation $\sum_i \sum_j$. Let $\sum_{c[i]=b}$ denote summation across $1 \le i \le \ell$ when $c[i] = b$. Let $\prod_j$ denote multiplication across $1 \le j \le m$.

Deriving from verifier steps 5 and 6:

$$
\begin{aligned}
t &= {g_1}^{t_0} \cdot \prod\nolimits_j \left({k_j}^{h_j \cdot t_j + E_1 \cdot F_j}\right) && \triangleright \text{ Verifier step 6} \\
&= {g_1}^{\sum (e_i \cdot F_j \cdot S_{i,j}) + \sum_{c[i]=0} \left(\sum_j \left(x^{e_i + f_j + s_{i,j}}\right)\right)} \cdot \prod\nolimits_j \left({k_j}^{h_j \cdot t_j + E_1 \cdot F_j}\right) && \triangleright \text{ Verifier step 5} \\
&= {g_1}^{\sum (e_i \cdot F_j \cdot S_{i,j}) + \sum_{c[i]=0} \left(\sum_j \left(x^{e_i + f_j + s_{i,j}}\right)\right)} && \triangleright \text{ Verifier steps 5.d.i and 5.d.ii} \\
&\quad \cdot \prod\nolimits_j \left({k_j}^{h_j \cdot \left(\sum_{c[i]=1} \left(x^{e_i + f_j + s_{i,j}}\right)\right) + \sum_{c[i]=1} (-e_i) \cdot F_j}\right) \\
&= {g_1}^{\sum_j \left(F_j \cdot \left(\sum_{c[i]=0} (e_i \cdot S_{i,j}) + \sum_{c[i]=1} (e_i \cdot S_{i,j})\right)\right) + \sum_{c[i]=0} \left(\sum_j \left(x^{e_i + f_j + s_{i,j}}\right)\right)} && \triangleright \text{ Split } e_i \cdot S_{i,j} \text{ cases} \\
&\quad \cdot \prod\nolimits_j \left({k_j}^{h_j \cdot \left(\sum_{c[i]=1} \left(x^{e_i + f_j + s_{i,j}}\right)\right) + \sum_{c[i]=1} (-e_i) \cdot F_j}\right) \\
&= {g_1}^{\sum_j \left(F_j \cdot \left(\sum_{c[i]=0} (e_i \cdot R_{i,j}) + \sum_{c[i]=1} (e_i \cdot (R_{i,j} + \gamma_j))\right)\right) + \sum_{c[i]=0} \left(\sum_j \left(x^{e_i + f_j + r_{i,j}}\right)\right)} && \triangleright \ S_{i,j} \text{ and } s_{i,j} \text{ from prover} \\
&\quad \cdot \prod\nolimits_j \left({k_j}^{h_j \cdot \left(\sum_{c[i]=1} \left(x^{e_i + f_j + r_{i,j} - \alpha_j}\right)\right) + \sum_{c[i]=1} (-e_i) \cdot F_j}\right) \\
&= {g_1}^{\sum_j \left(F_j \cdot \left(\sum_{c[i]=0} (e_i \cdot R_{i,j}) + \sum_{c[i]=1} (e_i \cdot (R_{i,j} + \gamma_j))\right)\right) + \sum_{c[i]=0} \left(\sum_j \left(x^{e_i + f_j + r_{i,j}}\right)\right)} && \triangleright \text{ Split } k_j \text{ exponent}
\end{aligned}
$$

$$\cdot \prod_j \left(k_j{}^{h_j \cdot \left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)} \cdot k_j{}^{\sum_{c[i]=1}(-e_i)\cdot F_j}\right)$$

$$= g_1{}^{\sum_j\left(F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot(R_{i,j}+\gamma_j))\right)\right)+\sum_{c[i]=0}\left(\sum_j\left(x^{e_i+f_j+r_{i,j}}\right)\right)}$$ ▹ k_j and h_j from statement

$$\cdot \prod_j \left(g_1{}^{x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)} \cdot g_1{}^{\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j}\right)$$

$$= g_1{}^{\sum_j\left(F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot(R_{i,j}+\gamma_j))\right)\right)+\sum_{c[i]=0}\left(\sum_j\left(x^{e_i+f_j+r_{i,j}}\right)\right)}$$ ▹ Merge g_1 exponent

$$\cdot \prod_j \left(g_1{}^{x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j}\right)$$

$$= g_1{}^{\left\{\begin{array}{c}\sum_j\left(F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot(R_{i,j}+\gamma_j))\right)\right)+\sum_{c[i]=0}\left(\sum_j\left(x^{e_i+f_j+r_{i,j}}\right)\right)\\ +\sum_j\left(x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j\right)\end{array}\right\}}$$ ▹ Merge g_1 exponent

$$= g_1{}^{\left\{\begin{array}{c}\sum_j\left(F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot(R_{i,j}+\gamma_j))\right)+\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)\right.\\ \left.+x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j\right)\end{array}\right\}}$$ ▹ Merge Σ_j cases

$$= g_1{}^{\sum_j\left(\begin{array}{c}F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot(R_{i,j}+\gamma_j))+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\right)\\ +\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)+x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)\end{array}\right)}$$ ▹ Combine F_j multiples

$$= g_1{}^{\sum_j\left(\begin{array}{c}F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot R_{i,j}+e_i\cdot\gamma_j+\gamma_j\cdot(-e_i))\right)\\ +\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)+x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)\end{array}\right)}$$ ▹ Merge $\Sigma_{c[i]=1}$ cases

$$= g_1{}^{\sum_j\left(\begin{array}{c}F_j\cdot\left(\sum_{c[i]=0}(e_i\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot R_{i,j})\right)+\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)\\ +x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)\end{array}\right)}$$ ▹ Cancellation of $e_i \cdot \gamma_j$

$$= g_1{}^{\sum_j\left(F_j\cdot\sum_i(e_i\cdot R_{i,j})+\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)+x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}-\alpha_j}\right)\right)\right)}$$ ▹ Collapse into conditionless Σ_i

$$= g_1{}^{\sum_j\left(F_j\cdot\sum_i(e_i\cdot R_{i,j})+\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)+x^{\alpha_j}\cdot\left(\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}}\cdot x^{-\alpha_j}\right)\right)\right)}$$ ▹ Split x exponent

$$= g_1{}^{\sum_j\left(F_j\cdot\sum_i(e_i\cdot R_{i,j})+\sum_{c[i]=0}\left(x^{e_i+f_j+r_{i,j}}\right)+\sum_{c[i]=1}\left(x^{e_i+f_j+r_{i,j}}\right)\right)}$$ ▹ Cancellation of x^{α_j}

$$= g_1{}^{\sum_j\left(F_j\cdot\sum_i(e_i\cdot R_{i,j})+\sum_i\left(x^{e_i+f_j+r_{i,j}}\right)\right)}$$ ▹ Collapse into conditionless Σ_i

$$= g_1{}^{\sum(e_i\cdot F_j\cdot R_{i,j})+\sum\left(x^{e_i+f_j+r_{i,j}}\right)}$$ ▹ Simplify using Σ definition

$$= t'$$ ▹ Prover step 7

Deriving from verifier steps 5 and 7:

$$T = g_2{}^{T_0} \cdot \prod_j \left(y_j{}^{E_2\cdot f_j} \cdot z_j{}^{-E_2\cdot F_j}\right)$$ ▹ Verifier step 7

$$= g_2{}^{\sum(e_i\cdot(f_j\cdot s_{i,j}+F_j\cdot S_{i,j}))}$$ ▹ Verifier steps 5.b and 5.d.iii

$$\cdot \prod_j \left(y_j{}^{\sum_{c[i]=1}(e_i)\cdot f_j} \cdot z_j{}^{\sum_{c[i]=1}(-e_i)\cdot F_j}\right)$$

$$= g_2{}^{\sum_j\left(\sum_{c[i]=0}(e_i\cdot(f_j\cdot s_{i,j}+F_j\cdot S_{i,j}))+\sum_{c[i]=1}(e_i\cdot(f_j\cdot s_{i,j}+F_j\cdot S_{i,j}))\right)}$$ ▹ Split cases for g_2 exponent

$$\cdot \prod_j \left(y_j{}^{\sum_{c[i]=1}(e_i)\cdot f_j} \cdot z_j{}^{\sum_{c[i]=1}(-e_i)\cdot F_j}\right)$$

$$= g_2^{\sum_j(\sum_{c[i]=0}(e_i\cdot(f_j\cdot r_{i,j}+F_j\cdot R_{i,j}))+\sum_{c[i]=1}(e_i\cdot(f_j\cdot(r_{i,j}-\alpha_j)+F_j\cdot(R_{i,j}+\gamma_j))))} \quad \triangleright\ S_{i,j} \text{ and } s_{i,j} \text{ from prover}$$
$$\cdot \prod_j (y_j^{\sum_{c[i]=1}(e_i)\cdot f_j} \cdot z_j^{\sum_{c[i]=1}(-e_i)\cdot F_j})$$
$$= g_2^{\sum_j(\sum_{c[i]=0}(e_i\cdot(f_j\cdot r_{i,j}+F_j\cdot R_{i,j}))+\sum_{c[i]=1}(e_i\cdot(f_j\cdot(r_{i,j}-\alpha_j)+F_j\cdot(R_{i,j}+\gamma_j))))}$$
$$\cdot \prod_j (g_2^{\alpha_j\cdot\sum_{c[i]=1}(e_i)\cdot f_j} \cdot g_2^{\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j}) \quad \triangleright\ y_j \text{ and } z_j \text{ from statement}$$
$$= g_2^{\sum_j(\sum_{c[i]=0}(e_i\cdot(f_j\cdot r_{i,j}+F_j\cdot R_{i,j}))+\sum_{c[i]=1}(e_i\cdot(f_j\cdot(r_{i,j}-\alpha_j)+F_j\cdot(R_{i,j}+\gamma_j))))} \quad \triangleright\ \text{Merge } g_2 \text{ exponent}$$
$$\cdot \prod_j (g_2^{\alpha_j\cdot\sum_{c[i]=1}(e_i)\cdot f_j+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j})$$
$$= g_2^{\left\{\begin{array}{c}\sum_j(\sum_{c[i]=0}(e_i\cdot(f_j\cdot r_{i,j}+F_j\cdot R_{i,j}))+\sum_{c[i]=1}(e_i\cdot(f_j\cdot(r_{i,j}-\alpha_j)+F_j\cdot(R_{i,j}+\gamma_j)))) \\ +\sum_j(\alpha_j\cdot\sum_{c[i]=1}(e_i)\cdot f_j+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j)\end{array}\right\}} \quad \triangleright\ \text{Merge } g_2 \text{ exponent}$$
$$= g_2^{\sum_j\left(\begin{array}{c}\sum_{c[i]=0}(e_i\cdot(f_j\cdot r_{i,j}+F_j\cdot R_{i,j})) \\ +\sum_{c[i]=1}(e_i\cdot(f_j\cdot(r_{i,j}-\alpha_j)+F_j\cdot(R_{i,j}+\gamma_j))) \\ +\alpha_j\cdot\sum_{c[i]=1}(e_i)\cdot f_j+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j\end{array}\right)} \quad \triangleright\ \text{Merge } \Sigma_j \text{ cases}$$
$$= g_2^{\sum_j\left(\begin{array}{c}\sum_{c[i]=0}(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j}) \\ +\sum_{c[i]=1}(e_i\cdot f_j\cdot r_{i,j}-e_i\cdot f_j\cdot\alpha_j+e_i\cdot F_j\cdot R_{i,j}+e_i\cdot F_j\cdot\gamma_j) \\ +\alpha_j\cdot\sum_{c[i]=1}(e_i)\cdot f_j+\gamma_j\cdot\sum_{c[i]=1}(-e_i)\cdot F_j\end{array}\right)} \quad \triangleright\ \text{Expand terms}$$
$$= g_2^{\sum_j\left(\begin{array}{c}\sum_{c[i]=0}(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j}) \\ +\sum_{c[i]=1}(e_i\cdot f_j\cdot r_{i,j}-e_i\cdot f_j\cdot\alpha_j+e_i\cdot F_j\cdot R_{i,j}+e_i\cdot F_j\cdot\gamma_j) \\ +\sum_{c[i]=1}(e_i\cdot f_j\cdot\alpha_j)+\sum_{c[i]=1}(-e_i\cdot F_j\cdot\gamma_j)\end{array}\right)} \quad \triangleright\ \text{Move terms into sums}$$
$$= g_2^{\sum_j\left(\begin{array}{c}\sum_{c[i]=0}(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j}) \\ +\sum_{c[i]=1}(e_i\cdot f_j\cdot r_{i,j}-e_i\cdot f_j\cdot\alpha_j+e_i\cdot F_j\cdot R_{i,j}+e_i\cdot F_j\cdot\gamma_j+e_i\cdot f_j\cdot\alpha_j-e_i\cdot F_j\cdot\gamma_j)\end{array}\right)} \quad \triangleright\ \text{Merge } \Sigma_{c[i]=1} \text{ cases}$$
$$= g_2^{\sum_j(\sum_{c[i]=0}(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j})+\sum_{c[i]=1}(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j}))} \quad \triangleright\ \text{Cancellations}$$
$$= g_2^{\sum_j(\sum_i(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j}))} \quad \triangleright\ \text{Collapse into conditionless } \Sigma_i$$
$$= g_2^{\sum(e_i\cdot f_j\cdot r_{i,j}+e_i\cdot F_j\cdot R_{i,j})} \quad \triangleright\ \text{Simplify using } \Sigma \text{ definition}$$
$$= T' \quad \triangleright\ \text{Prover step 7}$$

This shows that all of the inputs to H in verifier step 8 are identical to the inputs to H in prover step 8. Therefore, $c = c'$ and the verifier will accept the proof.

8.4.1.2 Proof of Zero Knowledge

The next task is to show that the Π_0 NIZKPK is statistically c-simulatable [Hen14, Def. 9] in the ROM. Concretely, there exists a simulator that takes as input the public values in the proof statement and a challenge c and, using a programmable random oracle H, produces a proof transcript with challenge c such that no PPT algorithm can distinguish this simulated transcript from an honestly generated transcript; the probability that any given transcript can

be distinguished from a real one is bounded by a negligible function of the security parameter, and so a PPT distinguisher would need to sample a super-polynomial number of transcripts. The existence of this simulator shows that the protocol is "zero-knowledge", because a PPT verifier cannot learn any information from an honest prover that it could not also learn from the simulator (without access to the witness).

The simulator for Π_0 can be constructed by simply choosing the responses uniformly at random from their allowable range, and then proceeding as an honest verifier would:

1. Take as input the public values in the statement (i.e., m, x, k_j, y_j, and z_j for $1 \leq j \leq m$) and the desired challenge c
2. Verify that $x \in \mathbb{G}_2$ and for each $1 \leq j \leq m$: $k_j \in \mathbb{G}_1$ and $y_j, z_j \in \mathbb{G}_2$ and abort otherwise
3. For $1 \leq i \leq \ell$ and $1 \leq j \leq m$: choose $s_{i,j} \xleftarrow{\$} \mathbb{Z}_{p_3}$ and $S_{i,j} \xleftarrow{\$} \mathbb{Z}_{2^{\ell'}}$
4. For each $1 \leq i \leq \ell$:
   a) If $c[i] = 0$ then set $u_i \leftarrow H_0(\text{"}\mathbb{G}\text{"} \| i \| s_{i,1} \| \ldots \| s_{i,m} \| S_{i,1} \| \ldots \| S_{i,m})$ and choose v_i to be a random value with the same bit-length as H_0's output space
   b) If $c[i] = 1$ then set $v_i \leftarrow H_1(\text{"}\mathbb{G}\text{"} \| i \| s_{i,1} \| \ldots \| s_{i,m} \| S_{i,1} \| \ldots \| S_{i,m})$ and choose u_i to be a random value with the same bit-length as H_1's output space
5. Compute $e_1, \ldots, e_\ell, f_1, \ldots, f_m, F_1, \ldots, F_m$ as in verifier step 3
6. Compute t_0, T_0, E_1, E_2 and t_j as in verifier step 5 (using c as the challenge)
7. Compute t as in verifier step 6
8. Compute T as in verifier step 7
9. Program the random oracle to output c given the input to H as in step 8
10. The simulated proof is $\pi = (c, s_{1,1}, \ldots, s_{\ell,m}, S_{1,1}, \ldots, S_{\ell,m}, w_1, \ldots, w_\ell)$

Note that the simulated $s_{i,j}$ values perfectly match the real distribution for an honest prover, since an honest prover chooses $r_{i,j}$ uniformly at random from the entire range modulo p_3.

The $S_{i,j}$ values are statistically indistinguishable from those sent by an honest prover. This can be shown by using a statistical result proven by Camenisch [Cam98]; this same result was previously referenced in Section 8.2.3 as justification for a choice of ℓ'.

Consider an honest prover with a witness $\gamma_j \in \mathbb{Z}_{2^\ell}$ and a value $R_{i,j}$ chosen uniformly at random from $\mathbb{Z}_{2^{\ell'}}$. If $c'[i] = 0$, then $S_{i,j} = R_{i,j}$; since this is independent of γ_j, the simulator perfectly matches this distribution and no distinguisher can possibly exist. If $c'[i] = 1$, then $S_{i,j} = R_{i,j} + \gamma_j$ (in $\mathbb{Z}$) and thus it is theoretically possible for a distinguisher to exist. There are three possible cases when $c'[i] = 1$:

1. $S_{i,j} < 2^\ell$
2. $2^\ell \leq S_{i,j} \leq 2^{\ell'}$
3. $S_{i,j} > 2^{\ell'}$

In the second case, there are exactly p_3 possible $(\gamma_j, R_{i,j})$ pairs that can produce any particular value for $S_{i,j}$. Consequently, the probability that $S_{i,j}$ takes on any value in the second case is uniform. The first and last cases both leak information about γ_j, because there are fewer than p_3 possible $(\gamma_j, R_{i,j})$ pairs that yield the values in these regions (in the extreme cases, γ_j is uniquely determined). Camenisch [Cam98, Th. 5.1] showed that the probability that $S_{i,j}$ ends up in these bad regions in at least one of n runs is upper-bounded by:

$$\Pr[S_{i,j} \notin [2^\ell, 2^{\ell'}]] \leq 1 - (1 - 2^{-(\ell'-\ell)+1})^n$$

For ℓ runs with $\ell = 128$ and $\ell' = 392$, this probability is upper-bounded by $2^{-\ell}$, which is negligible. Therefore, Π_0 is statistically c-simulatable for witnesses $0 < \gamma_j < p_3 < 2^\ell$.

8.4.1.3 Proof of Soundness

To show soundness, it must be shown that if an honest verifier accepts a proof, then with overwhelming probability the prover "knows" the correct witnesses. This idea is formalized by defining a rewinding extractor that can, given two responses for a proof with the same Fiat-Shamir [FS87] random oracle input, efficiently recover the associated witnesses. This excludes dishonest provers that can cheat with probability $2^{-\ell}$ by guessing the random oracle output when choosing the statement or their commitments. If the extractor functions for two random oracle outputs that are chosen uniformly at random, then soundness is proved for all other possible cheating provers.

Assume that an honest verifier has accepted the proof. Verifier step 1 ensures that x, k_j, y_j, and z_j for $1 \leq j \leq m$ are all valid group elements, and thus verifier steps 6 and 7 show that t

and T are valid group elements. From the group definitions, it is then possible to uniquely write:

$$k_j = g_1{}^{\mathbf{k_j}}$$
$$y_j = g_2{}^{\mathbf{y_j}}$$
$$z_j = g_2{}^{\mathbf{z_j}}$$
$$t = g_1{}^{\mathbf{t}}$$
$$T = g_2{}^{\mathbf{T}}$$

In other words, every element k_j and t have a unique discrete logarithm with respect to g_1 in $\mathbb{Z}_{p_2}$, and every element y_j, z_j, and T have a unique discrete logarithm with respect to g_2 in $\mathbb{Z}_{p_3}$ (although the prover may not necessarily know these discrete logarithms). These discrete logarithms are denoted using **boldface** font, as shown above.

The rewinding extractor runs the prover to produce a proof for a Fiat-Shamir challenge c_1, rewinds the prover to the point of random oracle invocation, and then resumes the prover with a different challenge c_2. The extractor then has access to two proofs with the same statement and commitments.

8.4.1.3.1 Notation

The soundness proof for Π_0 uses the following notation:

- Let $\sum_i$ denote summation across $1 \le i \le \ell$ and $\sum_j$ denote summation across $1 \le j \le m$.
- Let $\sum$ denote the combined summation $\sum_i \sum_j$.
- Let $\sum_{c[i]=b}$ denote summation across $1 \le i \le \ell$ when $c[i] = b$.
- Let $\prod_j$ denote multiplication across $1 \le j \le m$.
- Let $\sum_i^*$ denote summation across $1 \le i \le \ell$ for which $c_1[i] = c_2[i]$.
- Let $\sum_{c[i]=b}^*$ denote summation across $1 \le i \le \ell$ for which $c_1[i] = c_2[i] = b$.
- Let $\sum_{c_1[i]=1}^*$ denote summation across $1 \le i \le \ell$ for which $c_1[i] = 1$ and $c_2[i] = 0$.
- Let $\sum_{c_2[i]=1}^*$ denote summation across $1 \le i \le \ell$ for which $c_2[i] = 1$ and $c_1[i] = 0$.
- For values $1 \le i \le \ell$ where $c_1[i] \ne c_2[i]$:

- Denote responses from the run r where $c_r[i] = 0$ as $s_{i,j,0}$ and $S_{i,j,0}$.
- Denote responses from the run $r' = 1 - r$ where $c_{r'}[i] = 1$ as $s_{i,j,1}$ and $S_{i,j,1}$.

- For values $1 \leq i \leq \ell$ where $c_1[i] = c_2[i]$, denote the (identical) responses as $s_{i,j}$ and $S_{i,j}$.

When $c_1[i] = c_2[i]$, it must be the case that $s_{i,j,0} = s_{i,j,1}$ and $S_{i,j,0} = S_{i,j,1}$ for all $1 \leq j \leq m$. These equalities hold because $s_{i,j}$ and $S_{i,j}$ are used in verifier step 2 to determine u_i and v_j using hashes H_0 and H_1, which are in turn input into the computation of c in verifier step 8; any variation between the responses for positions i with $c_1[i] = c_2[i]$ would necessarily produce differing random oracle outputs, causing the verifier to fail—a contradiction.

8.4.1.3.2 Subtracting t Responses

Consider the value t computed in verifier step 6. When the verifier accepts an individual proof with challenge c, this ensures:

$$
\begin{aligned}
t &= {g_1}^{t_0} \cdot \prod_j ({k_j}^{h_j \cdot t_j + E_1 \cdot F_j}) && \triangleright \text{ Verifier step 6} \\
\mathbf{t} &= t_0 + \sum_j (\mathbf{k_j} \cdot (h_j \cdot t_j + E_1 \cdot F_j)) && \triangleright \text{ Discrete logarithm} \\
&= \sum_j (\sum_i (e_i \cdot F_j \cdot S_{i,j}) + \sum_{c[i]=0} (x^{e_i + f_j + s_{i,j}}) && \triangleright \text{ Verifier step 5} \\
&\qquad + \mathbf{k_j} \cdot (h_j \cdot \sum_{c[i]=1} (x^{e_i + f_j + s_{i,j}}) - \sum_{c[i]=1} (e_i) \cdot F_j))
\end{aligned}
$$

The verifier accepts the response for c_1, so the equation can be rewritten in that context. In anticipation of causing terms to cancel out, the summations can also be split into conditional summations that cover the same ranges:

$$
\begin{aligned}
\mathbf{t} = \sum_j (& \sum_i^* (e_i \cdot F_j \cdot S_{i,j}) + \sum_{c[i]=0}^* (x^{e_i + f_j + s_{i,j}}) \\
& + \mathbf{k_j} \cdot (h_j \cdot \sum_{c[i]=1}^* (x^{e_i + f_j + s_{i,j}}) - \sum_{c[i]=1}^* (e_i) \cdot F_j) \\
& + \sum_{c_1[i]=1}^* (e_i \cdot F_j \cdot S_{i,j,1}) + \sum_{c_2[i]=1}^* (e_i \cdot F_j \cdot S_{i,j,0}) + \sum_{c_2[i]=1}^* (x^{e_i + f_j + s_{i,j,0}}) \\
& + \mathbf{k_j} \cdot (h_j \cdot \sum_{c_1[i]=1}^* (x^{e_i + f_j + s_{i,j,1}}) - \sum_{c_1[i]=1}^* (e_i) \cdot F_j))
\end{aligned}
$$

Similarly, since the verifier accepts the response for c_2, the following equation holds:

$$
\begin{aligned}
\mathbf{t} = \sum_j (& \sum_i^* (e_i \cdot F_j \cdot S_{i,j}) + \sum_{c[i]=0}^* (x^{e_i + f_j + s_{i,j}}) \\
& + \mathbf{k_j} \cdot (h_j \cdot \sum_{c[i]=1}^* (x^{e_i + f_j + s_{i,j}}) - \sum_{c[i]=1}^* (e_i) \cdot F_j)
\end{aligned}
$$

$$+ \sum^{*}_{c_1[i]=1}(e_i \cdot F_j \cdot S_{i,j,0}) + \sum^{*}_{c_2[i]=1}(e_i \cdot F_j \cdot S_{i,j,1}) + \sum^{*}_{c_1[i]=1}(x^{e_i+f_j+s_{i,j,0}})$$
$$+ \mathbf{k_j} \cdot (h_j \cdot \sum^{*}_{c_2[i]=1}(x^{e_i+f_j+s_{i,j,1}}) - \sum^{*}_{c_2[i]=1}(e_i) \cdot F_j))$$

Subtracting the equation for the c_1 scenario from the equation for the c_2 scenario cancels out the terms for matching challenge bits, yielding the following equation:

$$\begin{aligned}0 = \sum_j (&\sum^{*}_{c_1[i]=1}(e_i \cdot F_j \cdot S_{i,j,0}) + \sum^{*}_{c_2[i]=1}(e_i \cdot F_j \cdot S_{i,j,1}) + \sum^{*}_{c_1[i]=1}(x^{e_i+f_j+s_{i,j,0}}) \\ &+ \mathbf{k_j} \cdot (h_j \cdot \sum^{*}_{c_2[i]=1}(x^{e_i+f_j+s_{i,j,1}}) - \sum^{*}_{c_2[i]=1}(e_i) \cdot F_j) \\ &- \sum^{*}_{c_1[i]=1}(e_i \cdot F_j \cdot S_{i,j,1}) - \sum^{*}_{c_2[i]=1}(e_i \cdot F_j \cdot S_{i,j,0}) - \sum^{*}_{c_2[i]=1}(x^{e_i+f_j+s_{i,j,0}}) \\ &- \mathbf{k_j} \cdot (h_j \cdot \sum^{*}_{c_1[i]=1}(x^{e_i+f_j+s_{i,j,1}}) - \sum^{*}_{c_1[i]=1}(e_i) \cdot F_j))\end{aligned}$$

The equation can be simplified as follows:

$$\begin{aligned}0 = \sum_j (&\sum^{*}_{c_1[i]=1}(e_i \cdot F_j \cdot S_{i,j,0} - e_i \cdot F_j \cdot S_{i,j,1}) && \triangleright \text{Merge cases} \\ &+ \sum^{*}_{c_2[i]=1}(e_i \cdot F_j \cdot S_{i,j,1} - e_i \cdot F_j \cdot S_{i,j,0}) \\ &+ \sum^{*}_{c_1[i]=1}(x^{e_i+f_j+s_{i,j,0}}) - \sum^{*}_{c_2[i]=1}(x^{e_i+f_j+s_{i,j,0}}) && \triangleright \text{Rearrange} \\ &+ \mathbf{k_j} \cdot (h_j \cdot \sum^{*}_{c_2[i]=1}(x^{e_i+f_j+s_{i,j,1}}) - \sum^{*}_{c_2[i]=1}(e_i) \cdot F_j && \triangleright \text{Factor out } \mathbf{k_j} \\ &\quad - h_j \cdot \sum^{*}_{c_1[i]=1}(x^{e_i+f_j+s_{i,j,1}}) + \sum^{*}_{c_1[i]=1}(e_i) \cdot F_j))\end{aligned}$$

$$\begin{aligned}0 = \sum_j (&F_j \cdot (\sum^{*}_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1})) && \triangleright \text{Factor out } F_j \\ &\quad + \sum^{*}_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0}))) \\ &+ x^{f_j} \cdot (\sum^{*}_{c_1[i]=1}(x^{e_i+s_{i,j,0}}) - \sum^{*}_{c_2[i]=1}(x^{e_i+s_{i,j,0}})) && \triangleright \text{Factor out } x^{f_j} \\ &+ \mathbf{k_j} \cdot (x^{f_j} \cdot h_j \cdot \sum^{*}_{c_2[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}}) && \triangleright \text{Factor out } x^{f_j} \\ &\quad - x^{f_j} \cdot h_j \cdot \sum^{*}_{c_1[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}}) \\ &\quad + \sum^{*}_{c_1[i]=1}(e_i) \cdot F_j - \sum^{*}_{c_2[i]=1}(e_i) \cdot F_j)) && \triangleright \text{Rearrange}\end{aligned}$$

$$\begin{aligned}0 = \sum_j (&F_j \cdot (\sum^{*}_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1})) \\ &\quad + \sum^{*}_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0}))) \\ &+ x^{f_j} \cdot (\sum^{*}_{c_1[i]=1}(x^{e_i+s_{i,j,0}}) - \sum^{*}_{c_2[i]=1}(x^{e_i+s_{i,j,0}}))\end{aligned}$$

$$
\begin{aligned}
&+ x^{f_j} \cdot \mathbf{k_j} \cdot (h_j \cdot \textstyle\sum^*_{c_2[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}}) && \triangleright \text{Factor out } x^{f_j} \\
&\qquad - h_j \cdot \textstyle\sum^*_{c_1[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}})) \\
&+ F_j \cdot \mathbf{k_j} \cdot (\textstyle\sum^*_{c_1[i]=1}(e_i) - \sum^*_{c_2[i]=1}(e_i))) && \triangleright \text{Factor out } F_j
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (F_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1})) && \triangleright \text{Merge } F_j \text{ terms} \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0})) \\
&+ \mathbf{k_j} \cdot (\textstyle\sum^*_{c_1[i]=1}(e_i) - \sum^*_{c_2[i]=1}(e_i))) \\
&+ x^{f_j} \cdot (\textstyle\sum^*_{c_1[i]=1}(x^{e_i+s_{i,j,0}}) - \sum^*_{c_2[i]=1}(x^{e_i+s_{i,j,0}}) && \triangleright \text{Merge } x^{f_j} \text{ terms} \\
&+ \mathbf{k_j} \cdot (h_j \cdot \textstyle\sum^*_{c_2[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}}) \\
&\qquad - h_j \cdot \textstyle\sum^*_{c_1[i]=1}(x^{e_i} \cdot x^{s_{i,j,1}})))
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (F_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1}) + \mathbf{k_j} \cdot e_i) && \triangleright \text{Merge } \textstyle\sum^*_{c_1[i]=1} \text{ case} \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0}) - \mathbf{k_j} \cdot e_i)) && \triangleright \text{Merge } \textstyle\sum^*_{c_2[i]=1} \text{ case} \\
&+ x^{f_j} \cdot (\textstyle\sum^*_{c_1[i]=1}(x^{e_i+s_{i,j,0}} - \mathbf{k_j} \cdot h_j \cdot x^{e_i} \cdot x^{s_{i,j,1}}) && \triangleright \text{Merge } \textstyle\sum^*_{c_1[i]=1} \text{ case} \\
&- \textstyle\sum^*_{c_2[i]=1}(\mathbf{k_j} \cdot h_j \cdot x^{e_i} \cdot x^{s_{i,j,1}} - x^{e_i+s_{i,j,0}}))) && \triangleright \text{Merge } \textstyle\sum^*_{c_2[i]=1} \text{ case}
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (F_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1} + \mathbf{k_j})) && \triangleright \text{Factor out } e_i \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0} - \mathbf{k_j}))) \\
&+ x^{f_j} \cdot (\textstyle\sum^*_{c_1[i]=1}(x^{e_i} \cdot (x^{s_{i,j,0}} - \mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}})) && \triangleright \text{Factor out } x^{e_i} \\
&- \textstyle\sum^*_{c_2[i]=1}(x^{e_i} \cdot (\mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}} - x^{s_{i,j,0}}))))
\end{aligned}
$$

However, it must be the case that every $\mathbf{y_j}$, $\mathbf{k_j}$, $s_{i,j,\star}$, and $S_{i,j,\star}$ value was chosen independently of every e_i, f_i, and F_j value. This must be the case because the e_i, f_j, and F_j values are computed in verifier step 3 from a hash H_e with input that depends on the other values: the hash depends on $\mathbf{y_j}$ and $\mathbf{k_j}$ because y_j and k_j are included in "$\mathbb{G}$" by definition and "$\mathbb{G}$" is an input to the hash; and the hash depends on every $s_{i,j,\star}$ and $S_{i,j,\star}$ value used in the above equation because every u_i and v_i value computed in verifier step 2 is also an input to the hash. The only way to make these variables dependent would be to predict the output of the hash—any such prediction would be incorrect with overwhelming probability. The e_i, f_j, and F_j values are drawn uniformly at

random from $[2, p_3)$ and are variously added and multiplied modulo p_3 to form $\ell \cdot m$ terms. The probability that any of these terms collide (due to the birthday attack) is $1 - (\ell \cdot m)! \cdot \binom{p_3}{\ell \cdot m} \cdot {p_3}^{-\ell \cdot m}$. It is easy to see that the probability of collisions is negligible in ℓ when m is set to a practical value (e.g., Safehouse typically sets $m \leq 10$). If Π_0 is used to prove a statement with an impractically large $m \geq \sqrt{p_3}/\ell$, then the probability that the $\ell \cdot m$ terms contain a collision becomes larger than $2^{-\ell}$. However, in this situation it also becomes overwhelmingly improbable that a malicious prover could correctly predict which terms collide: the probability of correctly predicting the location of colliding terms in this scenario is always less than $1/\sqrt{p_3} \approx 2^{-\ell}$. A proof created by a malicious prover will fail to verify if the prover mispredicts which terms will collide and chooses any corresponding values dishonestly, so a large value of m is not helpful to a malicious prover. Consequently, for any value of m, the simplified equation implies with overwhelming probability that the terms multiplied by F_j and x^{f_j} are both zero. The following sections discuss the implications of the terms being zero: it is possible to extract witnesses for the proof statement.

8.4.1.3.3 F_j Term in t Response

The F_j term in the simplified equation from Section 8.4.1.3.2 shows that:

$$0 = \sum\nolimits^{*}_{c_1[i]=1} (e_i \cdot (S_{i,j,0} - S_{i,j,1} + \mathbf{k_j})) + \sum\nolimits^{*}_{c_2[i]=1} (e_i \cdot (S_{i,j,1} - S_{i,j,0} - \mathbf{k_j}))$$

Since these summations are over disjoint sets, all e_i terms are distinct random values from $[2, p_3)$. As before, this implies that all terms multiplied by an e_i are zero with overwhelming probability. For every $1 \leq i \leq \ell$ where $c_1[i] = 1$, this implies:

$$0 = S_{i,j,0} - S_{i,j,1} + \mathbf{k_j}$$

Likewise, for every $1 \leq i \leq \ell$ where $c_2[i] = 1$, this implies:

$$0 = S_{i,j,1} - S_{i,j,0} - \mathbf{k_j}$$

In all cases, $\mathbf{k_j} = S_{i,j,1} - S_{i,j,0}$ when $c_1[i] \neq c_2[i]$.

8.4.1.3.4 x^{f_j} Term in t Response

The x^{f_j} term in the simplified equation from Section 8.4.1.3.2 shows that:

$$0 = \sum\nolimits^{*}_{c_1[i]=1} (x^{e_i} \cdot (x^{s_{i,j,0}} - \mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}})) - \sum\nolimits^{*}_{c_2[i]=1} (x^{e_i} \cdot (\mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}} - x^{s_{i,j,0}}))$$

For the same reason as before, the x^{e_i} values are distinct, large, and chosen independently of the inner terms. With overwhelming probability, this implies that for every $1 \leq i \leq \ell$ where $c_1[i] = 1$:

$$0 = x^{s_{i,j,0}} - \mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}}$$

Likewise, for every $1 \leq i \leq \ell$ where $c_2[i] = 1$, this implies:

$$0 = \mathbf{k_j} \cdot h_j \cdot x^{s_{i,j,1}} - x^{s_{i,j,0}}$$

In all cases, $h_j \cdot \mathbf{k_j} = x^{s_{i,j,0} - s_{i,j,1}}$ when $c_1[i] \neq c_2[i]$.

8.4.1.3.5 Extracting Witnesses from t

Section 8.4.1.3.3 and Section 8.4.1.3.4 showed that for all $1 \leq i \leq \ell$ where $c_1[i] \neq c_2[i]$, the extractor can with overwhelming probability extract the following witnesses:

- $\mathbf{k_j} = \gamma_j = S_{i,j,1} - S_{i,j,0}$;
- $\beta_j = x^{s_{i,j,0} - s_{i,j,1}}$; and
- $\alpha_j = s_{i,j,0} - s_{i,j,1} \pmod{p_3}$.

Moreover, verifier step 5.e ensures that $0 \leq S_{i,j,0} < 2^{\ell'+1}$ and $0 \leq S_{i,j,1} < 2^{\ell'+1}$. Therefore, the witness $\gamma_j = S_{i,j,1} - S_{i,j,0}$ is in the range $(-2^{\ell'+1}, 2^{\ell'+1})$ as required by the proof statement.

8.4.1.3.6 Subtracting T Responses

The next step is to consider the value T computed in verifier step 7. When the verifier accepts an individual proof with challenge c, this ensures:

$$T = {g_2}^{T_0} \cdot \prod_j ({y_j}^{E_2 \cdot f_j} \cdot {z_j}^{-E_2 \cdot F_j}) \qquad \triangleright \text{ Verifier step 7}$$

$$\mathbf{T} = T_0 + \sum_j (\mathbf{y_j} \cdot (E_2 \cdot f_j) + \mathbf{z_j} \cdot (-E_2 \cdot F_j)) \qquad \triangleright \text{ Discrete logarithm}$$

$$\mathbf{T} = \sum_j (\sum_i (e_i \cdot (f_j \cdot s_{i,j} + F_j \cdot S_{i,j})) + \mathbf{y_j} \cdot (\sum_{c[i]=1} (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c[i]=1} (e_i) \cdot F_j)) \qquad \triangleright \text{ Verifier step 5}$$

Since the verifier accepts the response for c_1, the equation can be rewritten in that context:

$$\begin{aligned}\mathbf{T} = \sum_j (&\sum_i^* (e_i \cdot (f_j \cdot s_{i,j} + F_j \cdot S_{i,j})) + \mathbf{y_j} \cdot (\sum_{c[i]=1}^* (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c[i]=1}^* (e_i) \cdot F_j) \\ &+ \sum_{c_1[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1})) + \sum_{c_2[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0})) \\ &+ \mathbf{y_j} \cdot (\sum_{c_1[i]=1}^* (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c_1[i]=1}^* (e_i) \cdot F_j))\end{aligned}$$

Similarly, since the verifier accepts the response for c_2, the following equation holds:

$$\begin{aligned}\mathbf{T} = \sum_j (&\sum_i^* (e_i \cdot (f_j \cdot s_{i,j} + F_j \cdot S_{i,j})) + \mathbf{y_j} \cdot (\sum_{c[i]=1}^* (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c[i]=1}^* (e_i) \cdot F_j) \\ &+ \sum_{c_1[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0})) + \sum_{c_2[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1})) \\ &+ \mathbf{y_j} \cdot (\sum_{c_2[i]=1}^* (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c_2[i]=1}^* (e_i) \cdot F_j))\end{aligned}$$

Subtracting the equation for the c_1 scenario from the equation for the c_2 scenario yields:

$$\begin{aligned}0 = \sum_j (&\sum_{c_1[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0})) + \sum_{c_2[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1})) \\ &+ \mathbf{y_j} \cdot (\sum_{c_2[i]=1}^* (e_i) \cdot f_j) + \mathbf{z_j} \cdot (-\sum_{c_2[i]=1}^* (e_i) \cdot F_j) \\ &- \sum_{c_1[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1})) - \sum_{c_2[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0})) \\ &- \mathbf{y_j} \cdot (\sum_{c_1[i]=1}^* (e_i) \cdot f_j) - \mathbf{z_j} \cdot (-\sum_{c_1[i]=1}^* (e_i) \cdot F_j))\end{aligned}$$

The equation can be simplified as follows:

$$\begin{aligned}0 = \sum_j (&\sum_{c_1[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0}) && \triangleright \text{Merge cases} \\ &\qquad - e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1})) \\ &+ \sum_{c_2[i]=1}^* (e_i \cdot (f_j \cdot s_{i,j,1} + F_j \cdot S_{i,j,1}) \\ &\qquad - e_i \cdot (f_j \cdot s_{i,j,0} + F_j \cdot S_{i,j,0})) \\ &+ f_j \cdot (\mathbf{y_j} \cdot \sum_{c_2[i]=1}^* (e_i) - \mathbf{y_j} \cdot \sum_{c_1[i]=1}^* (e_i)) && \triangleright \text{Factor out } f_j \\ &+ F_j \cdot (\mathbf{z_j} \cdot (-\sum_{c_2[i]=1}^* (e_i)) - \mathbf{z_j} \cdot (-\sum_{c_1[i]=1}^* (e_i)))) && \triangleright \text{Factor out } F_j\end{aligned}$$

$$\begin{aligned}0 = \sum_j (&\sum_{c_1[i]=1}^* (f_j \cdot (e_i \cdot (s_{i,j,0} - s_{i,j,1})) && \triangleright \text{Factor out } f_j \cdot e_i \\ &\qquad + F_j \cdot (e_i \cdot (S_{i,j,0} - S_{i,j,1}))) && \triangleright \text{Factor out } F_j \cdot e_i \\ &+ \sum_{c_2[i]=1}^* (f_j \cdot (e_i \cdot (s_{i,j,1} - s_{i,j,0})) && \triangleright \text{Factor out } f_j \cdot e_i\end{aligned}$$

$$
\begin{aligned}
&\quad + F_j \cdot (e_i \cdot (S_{i,j,1} - S_{i,j,0}))) && \triangleright \text{ Factor out } F_j \cdot e_i \\
&+ f_j \cdot (\mathbf{y_j} \cdot \textstyle\sum^*_{c_2[i]=1}(e_i) - \mathbf{y_j} \cdot \sum^*_{c_1[i]=1}(e_i)) \\
&+ F_j \cdot (\mathbf{z_j} \cdot (-\textstyle\sum^*_{c_2[i]=1}(e_i)) - \mathbf{z_j} \cdot (-\sum^*_{c_1[i]=1}(e_i))))
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (f_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (s_{i,j,0} - s_{i,j,1})) && \triangleright \text{ Factor out } f_j \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (s_{i,j,1} - s_{i,j,0})) \\
&+ \mathbf{y_j} \cdot \textstyle\sum^*_{c_2[i]=1}(e_i) - \mathbf{y_j} \cdot \sum^*_{c_1[i]=1}(e_i)) \\
&+ F_j \cdot (\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1})) && \triangleright \text{ Factor out } F_j \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0})) \\
&+ \mathbf{z_j} \cdot (-\textstyle\sum^*_{c_2[i]=1}(e_i)) - \mathbf{z_j} \cdot (-\sum^*_{c_1[i]=1}(e_i))))
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (f_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (s_{i,j,0} - s_{i,j,1}) - \mathbf{y_j} \cdot e_i) && \triangleright \text{ Merge } \textstyle\sum^*_{c_1[i]=1} \text{ case} \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (s_{i,j,1} - s_{i,j,0}) + \mathbf{y_j} \cdot e_i)) && \triangleright \text{ Merge } \textstyle\sum^*_{c_2[i]=1} \text{ case} \\
&+ F_j \cdot (\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1}) + \mathbf{z_j} \cdot e_i) && \triangleright \text{ Merge } \textstyle\sum^*_{c_1[i]=1} \text{ case} \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0}) - \mathbf{z_j} \cdot e_i))) && \triangleright \text{ Merge } \textstyle\sum^*_{c_2[i]=1} \text{ case}
\end{aligned}
$$

$$
\begin{aligned}
0 = \textstyle\sum_j (f_j \cdot (&\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (s_{i,j,0} - s_{i,j,1} - \mathbf{y_j})) && \triangleright \text{ Factor out } f_j \cdot e_i \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (s_{i,j,1} - s_{i,j,0} + \mathbf{y_j}))) \\
&+ F_j \cdot (\textstyle\sum^*_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1} + \mathbf{z_j})) \\
&+ \textstyle\sum^*_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0} - \mathbf{z_j}))))
\end{aligned}
$$

The same logic as in Section 8.4.1.3.3 can be applied to the simplified equation: since the f_j and F_j terms are large and must be independent of the prover's responses, with overwhelming probability all of the terms multiplied by these random coefficients must be zero to satisfy the equation. The following sections discuss how to extract witnesses based on the requirement that each term is zero.

8.4.1.3.7 f_j Term in T Response

The f_j term in the simplified equation from Section 8.4.1.3.6 shows that:

$$0 = \sum\nolimits^{*}_{c_1[i]=1}(e_i \cdot (s_{i,j,0} - s_{i,j,1} - \mathbf{y_j})) + \sum\nolimits^{*}_{c_2[i]=1}(e_i \cdot (s_{i,j,1} - s_{i,j,0} + \mathbf{y_j}))$$

Once again, these e_i terms are large, independent of the $s_{i,j,\star}$ values, and disjoint. With overwhelming probability, this implies that for every $1 \leq i \leq \ell$ where $c_1[i] = 1$:

$$0 = s_{i,j,0} - s_{i,j,1} - \mathbf{y_j}$$

Likewise, for every $1 \leq i \leq \ell$ where $c_2[i] = 1$, this implies:

$$0 = s_{i,j,1} - s_{i,j,0} + \mathbf{y_j}$$

In all cases, $\mathbf{y_j} = s_{i,j,0} - s_{i,j,1}$ when $c_1[i] \neq c_2[i]$.

8.4.1.3.8 F_j Term in T Response

The F_j term in the simplified equation from Section 8.4.1.3.6, shows that:

$$0 = \sum\nolimits^{*}_{c_1[i]=1}(e_i \cdot (S_{i,j,0} - S_{i,j,1} + \mathbf{z_j})) + \sum\nolimits^{*}_{c_2[i]=1}(e_i \cdot (S_{i,j,1} - S_{i,j,0} - \mathbf{z_j}))$$

With overwhelming probability, this implies that for every $1 \leq i \leq \ell$ where $c_1[i] = 1$:

$$0 = S_{i,j,0} - S_{i,j,1} + \mathbf{z_j}$$

Likewise, for every $1 \leq i \leq \ell$ where $c_2[i] = 1$, this implies:

$$0 = S_{i,j,1} - S_{i,j,0} - \mathbf{z_j}$$

In all cases, $\mathbf{z_j} = S_{i,j,1} - S_{i,j,0}$ when $c_1[i] \neq c_2[i]$.

8.4.1.3.9 Extracting Witnesses from T

Section 8.4.1.3.7 and Section 8.4.1.3.8 showed that for all $1 \leq i \leq \ell$ where $c_1[i] \neq c_2[i]$, the extractor can with overwhelming probability extract the following witnesses:

- $\mathbf{y_j} = s_{i,j,0} - s_{i,j,1} = \alpha_j \pmod{p_3}$; and

- $\mathbf{z_j} = S_{i,j,1} - S_{i,j,0} = \gamma_j \pmod{p_3}$.

8.4.1.3.10 Extractor Construction

When c_1 and c_2 differ, by definition they must differ in at least one bit position $1 \leq i \leq \ell$. Section 8.4.1.3.5 and Section 8.4.1.3.9 showed that the extractor can recover witnesses α_j, β_j, and γ_j for $1 \leq j \leq m$ from the responses $s_{i,\star,\star}$ and $S_{i,\star,\star}$ for any i such that $c_1[i] \neq c_2[i]$. The aforementioned sections also show that these witnesses satisfy all conditions in the proof statement defined in Section 8.2.7.

This extractor excludes the possibility of cheating provers that can successfully reply to two arbitrary but differing challenges for the same commitment. Therefore, a cheating prover can convince the verifier for at most one challenge for a given commitment. Since H is a random oracle, the probability that this challenge is chosen is negligible: $2^{-\ell}$.

8.4.2 Security of BRAKEM Construction

This section sketches a proof that the $\text{BRAKEM}^{\text{DDL}}_{\star}$ construction in Section 8.2.8 satisfies the security properties defined in Section 8.1.

8.4.2.1 Correctness, Public Verifiability, and Verifiability

Correctness of the scheme is straightforward to see, assuming that the NIZKPKs are correct. The correctness of the BDLEQ NIZKPKs in $\forall_{1\leq i\leq m}\pi_i$ are clear, and the correctness of Π_0 was shown in Section 8.4.1.1.

To show public verifiability, it must be shown that if a call to $\text{BRAKEM}^{\text{DDL}}_{x}$.Verify succeeds, then every public key set R_i is valid, all of the new public keys pk^*_i are valid, and any correct call to $\text{BRAKEM}^{\text{DDL}}_{x}$.Decapsulate would return the correct sk^*_i. If verification succeeds, then verification of π_0 and $\forall_{1\leq i\leq m}\pi_i$ has succeeded. Step 1 in the verification procedure for Π_0 (see Section 8.2.7) ensures that the burner public key x and all of the pk^*_i public keys are in $\mathbb{G}_2$. Step 1 in the verification procedure for BDLEQ (see Section 8.2.2.3), which succeeds for each π_i, ensures that $R_{i,j}/x$ is in $\mathbb{G}_2$ for all $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$. Since x is already confirmed to be in $\mathbb{G}_2$, this means that $R_{i,j}$ must also be in $\mathbb{G}_2$, and thus all R_i are valid. It remains to be shown that any correct $\text{BRAKEM}^{\text{DDL}}_{x}$.Decapsulate call would return the correct sk^*_i. The call to $\text{BRAKEM}^{\text{DDL}}_{x}$.Verify within $\text{BRAKEM}^{\text{DDL}}_{x}$.Decapsulate succeeds due to the premise, so the only requirement is that

$sk_i^* = {}^{y_i{}^{sk}}/_{h_{i,j}} \pmod{p_3}$ for sk corresponding to $R_{i,j} = {g_2}^{sk}$. Since π_0 is valid, the definition of the proof statement for Π_0 (see Section 8.2.7) ensures that this equality holds.

To show verifiability, it must be shown that if every public key set R_i is valid and a correct call to BRAKEM$_x^{\text{DDL}}$.Decapsulate outputs sk_i^* corresponding to pk_i^*, then all of the new public keys are valid and the ciphertexts and proofs could have been created by a call to BRAKEM$_x^{\text{DDL}}$.Encapsulate. Since the BRAKEM$_x^{\text{DDL}}$.Decapsulate call did not output $\perp$, the BRAKEM$_x^{\text{DDL}}$.Verify call within must have succeeded, and thus the proofs π_0 and $\forall_{1\leq i\leq m}\pi_i$ verified successfully. Π_0 ensures that all of the new public keys $\forall_{1\leq i\leq m}pk_i^*$ are valid, so this immediately satisfies the first requirement. For the second requirement, the random coins s' passed to BRAKEM$_x^{\text{DDL}}$.Encapsulate are only used for three purposes: generating sk_i^* for all i, generating the ElGamal secrets r_i for all i, and generating the randomness for the NIZKPKs. Since all of the pk_i^* are valid and therefore must have discrete logarithms in $\mathbb{G}_2$ with respect to g_2, there must exist corresponding private keys that could be derived from s'. The same argument holds $\forall y_i = {g_2}^{r_i}$, since Π_0 ensures that each $y_i \in \mathbb{G}_2$ and thus there must exist ElGamal secrets that could be derived from s'. Since all of the NIZKPKs are valid, the soundness of the proof systems ensures that there exists randomness derived from s' that could produce π_0 and $\forall_{1\leq i\leq m}\pi_i$ with overwhelming probability.

8.4.2.2 Key Secrecy

The most challenging property to prove for BRAKEM$_\star^{\text{DDL}}$ is the key secrecy property. While the key secrecy proof for BRAKEM$_\star^{\text{2DDL}}$ requires only standard assumptions, the unusual use of ElGamal in BRAKEM$_\star^{\text{DDL}}$ requires unusual hardness assumptions. Section 8.4.2.2.1 defines the relevant assumptions, and the security proof is sketched in Sections 8.4.2.2.2, 8.4.2.2.3, and 8.4.2.2.4.

8.4.2.2.1 Hardness Assumptions

Proving the security of BRAKEM$_\star^{\text{DDL}}$ requires two new hardness assumptions. The first required assumption is the *discrete logarithm hidden subgroup* assumption:

Definition 1 | The discrete logarithm hidden subgroup assumption

Given a multiplicative subgroup $\mathbb{G}_2$ of order p_3 in $\mathbb{Z}_{p_2}^*$ with primes p_3, p_2, and q such that $p_2 = 2 \cdot p_3 \cdot q + 1$, a group $\mathbb{G}_1$ of order p_2, and elements $g_1 \in \mathbb{G}_1$ and $\mu \in \mathbb{G}_1$, it is computationally infeasible for a PPT adversary to decide if $\text{dlog}_{g_1}(\mu) \in \mathbb{G}_2$ with non-negligible advantage. (Refs: 265 and 503)

It would be quite surprising if the discrete logarithm hidden subgroup assumption did not hold in the groups defined in Section 8.2.1. Since the DDH assumption is assumed to be hard in

$\mathbb{G}_1$, it is typical to assume that an adversary can only perform a few select operations "in the exponent": multiplying by a constant (by exponentiating μ), adding a value (by multiplying μ with another $\mathbb{G}_1$ element), or brute-force testing for discrete logarithms from a small subset of possibilities with negligible size with respect to the security parameter. It is not clear how these operations could be used to construct a membership test for $\mathbb{G}_2$ "in the exponent". Typically, a value $v \in [1, p_2)$ is tested for membership in $\mathbb{G}_2$ by checking $v^{p_3} = 1 \pmod{p_2}$. This method cannot be used because modular exponentiation would require modular multiplication, but this cannot be done "in the exponent" without already knowing $\text{dlog}_{g_1}(\mu)$. A successful adversary would require a different method. These reasons are similar to the reason that the DDL problem is hard in groups where the DL problem is hard.

The second new assumption is more unusual. Given the groups defined in Section 8.2.1 and a value $x \in [1, p_2)$, let $\Phi(x)$ compute the component of x belonging to the multiplicative subgroup of order $2q$. Specifically:

$$\Phi(x) = x^{p_3(p_3^{-1} \pmod{2q})} \pmod{p_2}$$

The *multiplicative subgroup rounding with verifiers* assumption is defined as follows:

Definition 2 | The multiplicative subgroup rounding with verifiers assumption

Given the values $\Phi(x)$, ${g_1}^x$, and ${g_2}^x$, it is computationally infeasible for a PPT adversary to distinguish the world where x was sampled uniformly at random from $[0, p_3)$ from the world where x was sampled uniformly at random from $[0, p_2 p_3)$. (Refs: 263 and 503)

The difficulty of the multiplicative subgroup rounding with verifiers problem is not immediately obvious. However, there are good reasons to suspect that the problem is hard. It is strongly related to the *multiplicative subgroup rounding* problem defined by Boneh et al. [BJN00, §2]:

Definition 3 | The multiplicative subgroup rounding problem

Let $z \in \mathbb{G}_2$ and $\Delta \in [1, 2^m)$. Given $u = z \cdot \Delta \pmod{p_2}$, find z.

When the order of $\mathbb{G}_2$ is significantly smaller than p_2, Δ is uniquely determined with high probability (this is true in the BRAKEM$_\star^{\text{DDL}}$ setting, where q is extremely large). Boneh et al. introduced the multiplicative subgroup rounding problem in order to demonstrate that when m is small, ElGamal encryption of non-group elements is insecure. They presented several algorithms that can recover z with a time complexity of $O(2^{m/2})$. These attacks are not practical when 2^m becomes cryptographically large, as is the case in BRAKEM$_\star^{\text{DDL}}$ where $m \approx 256$. In these settings, the known algorithms that solve the multiplicative subgroup rounding problem have the same asymptotic running time as generic group attacks against the DL problem in terms of m. Consequently, undermining the security of BRAKEM$_\star^{\text{DDL}}$ would require the development of

asymptotically superior algorithms for solving the multiplicative subgroup rounding with verifiers problem.

It is relatively easy to see that the multiplicative subgroup rounding problem is difficult in a generic group model with the parameters used by BRAKEM$^{\text{DDL}}_{\star}$. Consider the case where the subgroup $\mathbb{G}_2$ is distributed within $[1, p_2)$ uniformly at random, rather than being a multiplicative subgroup, and the adversary is given one oracle to test for subgroup membership and a second oracle to generate a subgroup member uniformly at random. The only attacks available to such an adversary are brute-forcing Δ and calling the membership testing oracle for $u \cdot \Delta^{-1}$, or generating a random group element z and checking to see if u/z is in $[1, 2^m)$, both of which are infeasible when m is large. Consequently, a successful attack would require taking advantage of the structure of the multiplicative subgroup. To the best of the author's knowledge, there is no known algorithm that can exploit the subgroup structure in this way.

Consider a variant of the multiplicative subgroup rounding with verifiers problem in which only $\Phi(x)$ is provided (i.e., ${g_1}^x$ and ${g_2}^x$ are omitted). This variant is essentially a decisional version of the multiplicative subgroup rounding problem: given an element H from the multiplicative subgroup of order $2q$ in $\mathbb{Z}^*_{p_2}$, decide if $H = \Phi(x)$ for $x \in [0, 2^m)$. The equivalence is that $p_3 \approx 2^m$, $H = u \cdot z^{-1}$, and $x = \Delta$. Since p_3 is cryptographically large, this is good reason to believe that this problem variant is hard.

It remains to be considered how the aforementioned variant relates to the overall multiplicative subgroup rounding with verifiers problem. The availability of ${g_1}^x$ and ${g_2}^x$ certainly grants additional tools to the adversary. At a minimum, these group elements act as verification oracles modulo their respective group orders. Concretely, the adversary can check to see if a candidate solution $x' = x \pmod{p_2}$ by checking if ${g_1}^{x'} = {g_1}^x$. Likewise, it can check if $x' = x \pmod{p_3}$ by checking if ${g_2}^{x'} = {g_2}^x$. This information cannot be learned from $\Phi(x)$ alone. It is possible that more information could be learned from the group elements, but due to the hardness of DDH in both individual groups, any extra information revealed should not be helpful. It is not clear how adversaries could make use of these extra verification oracles. The structure of the multiplicative subgroup is unlikely to be helpful for the same reason that the multiplicative subgroup rounding problem seems to be difficult.

8.4.2.2.2 Proof Overview

The key secrecy security theorem for BRAKEM$^{\text{DDL}}_{\star}$ can be expressed in terms of the hardness assumptions given in Section 8.4.2.2.1:

Theorem 7 | $\text{BRAKEM}_{\star}^{\text{DDL}}$ is secure

If the DDH problem is hard in $\mathbb{G}_2$, the DL problem is hard in $\mathbb{G}_1$, and the discrete logarithm hidden subgroup and multiplicative subgroup rounding with verifiers problems are hard for the groups, then $\text{BRAKEM}_{\star}^{\text{DDL}}$ provides BRAKEM key secrecy (as defined in Section 8.1). (Refs: 260 [a] [b], 267, and 503)

The key secrecy game for BRAKEM is complex, generating many different group elements and NIZKPKs. To prove Theorem 7, it is necessary to tame this complexity. One way to do this is to express the proof as a sequence of games [Sho04]. The first game in the sequence is played by the actual challenger (i.e., $\text{BRAKEM}_{\star}^{\text{DDL}}$), and the last game is trivially shown to be secure. By showing that each game transition is indistinguishable (or else an efficient adversary could be constructed to attack a problem that is presumed to be hard), the proof demonstrates that an adversary attacking $\text{BRAKEM}_{\star}^{\text{DDL}}$ has negligible advantage compared to an adversary attacking the trivially secure scheme. This is a well known proof technique for complex cryptographic primitives and protocols.

Proving Theorem 7 requires a two-step approach. First, Section 8.4.2.2.3 uses a sequence of games to show key secrecy for a specific new public key when all of the recipient public keys in the call to $O_{\text{Encapsulate}}$ can be controlled by the reduction. Given this result, Section 8.4.2.2.4 shows how to prove key secrecy for the whole construction.

8.4.2.2.3 Encapsulation Game Sequence

This section uses a sequence of games to prove an intermediate step. Consider the following game:

Game 0

1. The challenger chooses $R_1, \ldots, R_m$, where each R_i is a set of public keys in $\mathbb{G}_2$. These values are sent to the adversary.
2. The challenger executes $\text{BRAKEM}_{x}^{\text{DDL}}.\text{Encapsulate}(R_1, \ldots, R_m)$. The adversary is sent the new public keys $pk_1^*, \ldots, pk_m^*$, ciphertexts $C_1, \ldots, C_m$, and the proof π.
3. The adversary outputs an index $1 \leq i \leq m$ and a guess sk'. The adversary wins if and only if $sk' = sk_i^*$.

This game can be viewed as a specific case within the overall BRAKEM key secrecy game defined in Section 8.1 played against $\text{BRAKEM}_{x}^{\text{DDL}}$ with burner public key x: the case when the adversary

calls $O_{\text{Encapsulate}}$ with recipient public keys that have been defined by the challenger. After showing that the adversary's advantage in this game is negligible, this result can be used in the overall key secrecy proof. In this initial game, the encapsulation operates exactly as defined in Section 8.2.8.

Game 1

This is the same as the previous game, except that all of the NIZKPKs are replaced by simulations. Specifically, in $\text{BRAKEM}_x^{\text{DDL}}$.Encapsulate, π_0 is replaced by a simulated version of Π_0, and each π_i for $1 \leq i \leq m$ is replaced by a simulated version of BDLEQ.

The security proofs showing that the NIZKPKs are zero-knowledge define the procedures for the simulations. Henry [Hen14, §3.2.3] sketches a simulator for the interactive form of the BDLEQ construction. This simulator can be transformed into one for the non-interactive form using the standard technique for Fiat-Shamir [FS87] proofs: programming the random oracle to return the selected challenge c for a commitment calculated by the simulator. A simulator for Π_0 is defined in Section 8.4.1.2.

Since any PPT adversary's advantage for distinguishing between simulated and real proofs is negligible for both NIZKPKs, any PPT adversary also has negligible advantage distinguishing Game 0 from Game 1.

Game 2

This game is the same as Game 1 except that $\text{BRAKEM}_x^{\text{DDL}}$.Encapsulate is modified to produce ElGamal ciphertexts using a random $\mathbb{G}_2$ element instead of the DH shared secret. The new encapsulation process operates as follows:

for each $(1 \leq i \leq m)$ {
  Generate key pair $(pk_i^*, sk_i^*) \leftarrow \text{BRAKEM}_x^{\text{DDL}}.\text{KeyGen}()$.
  Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
  Compute $y_i \leftarrow {g_2}^{r_i}$.
  Compute $\overline{sk_i^*} \leftarrow sk_i^{*-1} \pmod{p_2}$.
  Choose $S_i \in \mathbb{G}_2$ uniformly at random.
  Compute $h_i \leftarrow S_i \times \overline{sk_i^*} \pmod{p_2}$. ▷ Replaced x^{r_i} with new variable S_i
  for each $(1 \leq j \leq |R_i|)$ {
    Choose $S_{i,j} \in \mathbb{G}_2$ uniformly at random.
    Compute $h_{i,j} \leftarrow S_{i,j} \times \overline{sk_i^*} \pmod{p_2}$. ▷ Replaced ${R_{i,j}}^{r_i}$ with new variable $S_{i,j}$
  }
  Compute $k_i \leftarrow {g_1}^{sk_i^*}$.
  Simulate π_i.

Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.

It must be shown that distinguishing Game 2 from Game 1 is at least as hard as the DDH problem in $\mathbb{G}_2$, which is assumed to be hard. Given a PPT adversary $\mathcal{A}$ that can distinguish the games with non-negligible advantage, it is possible to construct a PPT adversary that solves the DDH problem with the same advantage. Given an input problem instance (A, B, C), the new adversary must return "1" if and only if $C = {g_2}^{\mathrm{dlog}_{g_2}(A)\cdot\mathrm{dlog}_{g_2}(B)}$. The new adversary runs $\mathcal{A}$, but modifies the challenger as follows:

Choose $\alpha_{i,j} \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Choose m and $|R_i|$ to send to $\mathcal{A}$ as usual, but set $R_{i,j} \leftarrow A^{\alpha_{i,j}}$. ▷ Blinded A as recipient
for each $(1 \le i \le m)$ {
Generate key pair $(pk_i^*, sk_i^*) \leftarrow \mathrm{BRAKEM}_x^{\mathrm{DDL}}.\mathrm{KeyGen}()$.
Choose $\beta_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Compute $y_i \leftarrow B^{\beta_i}$. ▷ Blinded B as ElGamal public key
Choose $\gamma_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Compute $\overline{sk_i^*} \leftarrow {sk_i^*}^{-1} \pmod{p_2}$.
Compute $S_i \leftarrow C^{\gamma_i}$.
Compute $h_i \leftarrow S_i \times \overline{sk_i^*} \pmod{p_2}$. ▷ Blinded C as DH shared secret
for each $(1 \le j \le |R_i|)$ {
Choose $\gamma_{i,j} \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Compute $S_{i,j} \leftarrow C^{\gamma_{i,j}}$.
Compute $h_{i,j} \leftarrow S_{i,j} \times \overline{sk_i^*} \pmod{p_2}$.
}
Compute $k_i \leftarrow {g_1}^{sk_i^*}$.
Simulate π_i.
Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.
Send $(pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$ to $\mathcal{A}$.
if ($\mathcal{A}$ outputs "Game 1") **return** "1".
if ($\mathcal{A}$ outputs "Game 2") **return** "0".

This procedure behaves identically to Game 1 when $C = {g_2}^{\text{dlog}_{g_2}(A)\cdot\text{dlog}_{g_2}(B)}$, so $\mathcal{A}$ returns "Game 1" with its usual advantage. In the other case, the values in the modified game are distributed identically to Game 2, so $\mathcal{A}$ returns "Game 2" with its usual advantage.

Game 3

This game is the same as Game 2 except that the private keys are generated from a larger range. In Game 2, each new key pair is generated as follows as part of $\text{BRAKEM}_x^{\text{DDL}}$.KeyGen:

Choose $sk_i^* \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Compute $pk_i^* \leftarrow {g_2}^{sk_i^*}$.

In Game 3, sk_i^* is chosen from $[0, p_2p_3)$ instead.

It must be shown that distinguishing Game 2 from Game 3 is at least as hard as the multiplicative subgroup rounding with verifiers problem (see Definition 2), which is assumed to be hard. Given a PPT adversary $\mathcal{A}$ that can distinguish the games with non-negligible advantage, it is possible to construct a PPT adversary that solves the multiplicative subgroup rounding with verifiers problem with the same advantage. Given an input problem instance $\phi = \Phi(z)$, $\mu_1 = {g_1}^z$, and $\mu_2 = {g_2}^z$, the new adversary runs $\mathcal{A}$, but modifies the challenger to behave as follows:

for each $(1 \le i \le m)$ {
  Assign $pk_i^* \leftarrow \mu_2$. ▷ μ_2 verifier as the new public key
  Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
  Compute $y_i \leftarrow {g_2}^{r_i}$. ▷ Normal ElGamal public key generation
  Choose $S_i \in \mathbb{G}_2$ uniformly at random.
  Compute $h_i \leftarrow S_i \cdot \phi^{-1}$. ▷ Ciphertext with ϕ as order-$2q$ component
  for each $(1 \le j \le |R_i|)$ {
    Choose $S_{i,j} \in \mathbb{G}_2$ uniformly at random.
    Compute $h_{i,j} \leftarrow S_{i,j} \cdot \phi^{-1}$. ▷ Ciphertext with ϕ as order-$2q$ component
  }
  Assign $k_i \leftarrow \mu_1$. ▷ μ_1 verifier as the $\mathbb{G}_1$ public key
  Simulate π_i.
  Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.
Send $(pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$ to $\mathcal{A}$.

if ($\mathcal{A}$ outputs "Game 2") **return** "World $[0, p_3)$".
if ($\mathcal{A}$ outputs "Game 3") **return** "World $[0, p_2 p_3)$".

Let $\Psi(z)$ represent the $\mathbb{G}_2$ component of z, just like $\Phi(z)$ represents the component from the order-$2q$ subgroup. Concretely:

$$\Psi(z) = z^{2q((2q)^{-1} \pmod{p_3})} \pmod{p_2}$$

Any element can be written as $z = \Psi(z)\Phi(z) \pmod{p_2}$. It is now easy to see that the modified encapsulation algorithm above produces values with the same distribution as either Game 2 or Game 3:

$$\begin{aligned} h_i &= S_i \cdot \phi^{-1} \\ h_i &= S_i \cdot (\Phi(z))^{-1} \\ &= S_i \cdot \Psi(z) \cdot (\Psi(z))^{-1} \cdot (\Phi(z))^{-1} \\ &= S_i \cdot \Psi(z) \cdot z^{-1} \end{aligned}$$

Since $\Psi(z) \in \mathbb{G}_2$ and S_i is sampled from $\mathbb{G}_2$ uniformly at random, this implies that $S_i \cdot \Psi(z)$ is also uniformly distributed in $\mathbb{G}_2$. Consequently, h_i, k_i, and pk_i^* correctly correspond to the case where $sk_i^* = z$. A similar calculation works for the $h_{i,j}$ ciphertexts. pk_i^* is identically distributed in both cases, because the uniform distribution over $[0, p_3)$ is the same as the uniform distribution over $[0, p_2 p_3)$ when taken modulo p_3. Theoretically, k_i could be distributed differently when $\mathrm{dlog}_{g_1}(k_1)$ is in $[0, p_3)$ instead of $[0, p_2 p_3)$. However, distinguishing between these distributions would be the same as determining whether or not the high bits of the discrete logarithm are zero—in the group setting for BRAKEM$^{\mathrm{DDL}}_{\star}$, this would mean deciding whether roughly 2816 bits are zero. It is known[31] that this problem is hard when the DL problem is hard in $\mathbb{G}_1$, so it is computationally infeasible for $\mathcal{A}$ to notice a change in the distribution of k_i. $\mathcal{A}$ therefore guesses the correct range for $sk_i^* = z$ with its usual advantage, and so the new adversary inherits the advantage for attacking multiplicative subgroup rounding with verifiers.

Game 4

This game is the same as Game 3 except that h_i is sampled uniformly at random from $[0, p_2)$, and the $h_{i,j}$ values are adjusted accordingly. The new encapsulation process operates as follows:

[31] Håstad and Näslund [HN04, Th. 10.2] provide one proof of this statement. Others have proven similar statements about the difficulty of learning groups of discrete logarithm bits.

for each $(1 \leq i \leq m)$ {
  Choose $sk_i^* \xleftarrow{\$} [0, p_2 p_3)$ uniformly at random. ▷ Key generation as in Game 3
  Compute $pk_i^* \leftarrow {g_2}^{sk_i^*}$.
  Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
  Compute $y_i \leftarrow {g_2}^{r_i}$.
  Choose $h_i \in [0, p_2)$ uniformly at random. ▷ Replaced $S_i \times \overline{sk_i^*} \pmod{p_2}$
  for each $(1 \leq j \leq |R_i|)$ {
    Choose $S_{i,j} \in \mathbb{G}_2$ uniformly at random.
    Compute $h_{i,j} \leftarrow S_{i,j} \cdot h_i$. ▷ Replaced $S_{i,j} \times \overline{sk_i^*} \pmod{p_2}$
  }
  Compute $k_i \leftarrow {g_1}^{sk_i^*}$.
  Simulate π_i.
  Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.

It must be shown that distinguishing Game 4 from Game 3 is at least as hard as the discrete logarithm hidden subgroup problem (see Definition 1), which is assumed to be hard. Given a PPT adversary $\mathcal{A}$ that can distinguish the games with non-negligible advantage, it is possible to construct a PPT adversary that solves the discrete logarithm hidden subgroup problem with the same advantage. Given an input problem instance $\mu \in \mathbb{G}_1$, the new adversary runs $\mathcal{A}$, but modifies the challenger to behave as follows:

for each $(1 \leq i \leq m)$ {
  Compute $pk_i^* \in \mathbb{G}_2$ uniformly at random.
  Compute $y_i \in \mathbb{G}_2$ uniformly at random.
  Choose ν in the multiplicative subgroup of order $2q$ in $\mathbb{Z}_{p_2}$ uniformly at random.
  Choose $S_i \in \mathbb{G}_2$ uniformly at random.
  Compute $h_i \leftarrow S_i \cdot \nu$.
  for each $(1 \leq j \leq |R_i|)$ {
    Choose $S_{i,j} \in \mathbb{G}_2$ uniformly at random.
    Compute $h_{i,j} \leftarrow S_{i,j} \cdot \nu$.
  }
  Compute $k_i \leftarrow \mu^{\nu}$. ▷ Blinded μ as the $\mathbb{G}_1$ public key
  Simulate π_i.

Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.
Send $(pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$ to $\mathcal{A}$.
if ($\mathcal{A}$ outputs "Game 3") **return** "1".
if ($\mathcal{A}$ outputs "Game 4") **return** "0".

Let $\alpha = \text{dlog}_{g_1}(\mu)$. In the case where $\alpha \in \mathbb{G}_2$, the values sent to $\mathcal{A}$ are identically distributed to Game 3. Consider the value h_i:

$$\begin{aligned} h_i &= S_i \cdot \nu \\ &= S_i \cdot \alpha^{-1} \cdot \alpha \cdot \nu \\ &= (S_i \cdot \alpha^{-1}) \cdot (\alpha \cdot \nu) \end{aligned}$$

A similar equality holds for each $h_{i,j}$ with respect to $S_{i,j}$. Next, consider the value k_i:

$$\begin{aligned} k_i &= \mu^{\nu} \\ &= {g_1}^{\alpha \cdot \nu} \end{aligned}$$

Note that since $\alpha \in \mathbb{G}_2$, these values are all identically distributed to Game 3 for the case where $sk_i^* = \alpha \cdot \nu + \delta p_2$ for some $\delta < p_3$ such that $sk_i^* = \text{dlog}_{g_2}(pk_i^*) \pmod{p_3}$. An appropriate value for δ is guaranteed to exist due to the Chinese remainder theorem.

In the case where $\alpha \notin \mathbb{G}_2$, the game is identically distributed to Game 4. There still exists a value $sk_i^* = \alpha \cdot \nu + \delta p_2$ that matches the distribution of Game 4 for k_i and pk_i^*. However, the h_i and $h_{i,j}$ values will be uniformly random values in $[0, p_2)$ that do not correspond to this sk_i^*, which is exactly the case in Game 4.

Game 5

This game is the same as Game 4 except that k_i is replaced by an element of $\mathbb{G}_1$ sampled uniformly at random, and sk_i^* is once again sampled uniformly from $[0, p_3)$. The resulting game is as follows:

for each $(1 \leq i \leq m)$ {
Choose $sk_i^* \xleftarrow{\$} [0, p_3)$ uniformly at random. ▷ Now sampled from $[0, p_3)$
Compute $pk_i^* \leftarrow {g_2}^{sk_i^*}$.

Choose $r_i \xleftarrow{\$} \mathbb{Z}_{p_3}$ uniformly at random.
Compute $y_i \leftarrow {g_2}^{r_i}$.
Choose $h_i \in [0, p_2)$ uniformly at random.
for each $(1 \leq j \leq |R_i|)$ {
    Choose $S_{i,j} \in \mathbb{G}_2$ uniformly at random.
    Compute $h_{i,j} \leftarrow S_{i,j} \cdot h_i$.
}
Choose $k_i \in \mathbb{G}_1$ uniformly at random. ▷ Replaced ${g_1}^{sk_i^*}$ with a random value
Simulate π_i.
Assign $C_i = (y_i, h_{i,1}, \ldots, h_{i,|R_i|})$.
}
Simulate π_0.
Assign $\pi = (\pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.

The k_i and pk_i^* values in the game are identically distributed to Game 4, because they always correspond to some other $sk' \in [0, p_2 p_3)$ due to the Chinese remainder theorem. Consequently, any adversary that attacks Game 5 can also attack Game 4 with equal advantage.

Summary

It is clear to see that any PPT adversary attacking Game 5 has negligible advantage. The game is trivially reduced to the DL problem in $\mathbb{G}_2$ by setting pk_i^* to the problem instance and randomly generating the other values, which are fully decoupled from $\text{dlog}_{g_2}(pk_i^*)$. As a result of the indistinguishability between each step of the game sequence, any adversary attacking Game 0 also has negligible advantage.

8.4.2.2.4 Overall Game Sequence

Given the intermediate result from Section 8.4.2.2.3, it is now possible to prove Theorem 7. Consider the key secrecy game as defined in Section 8.1. The adversary is given access to three oracles: O_{KeyGen} generates new public keys, $O_{\text{Encapsulate}}$ encapsulates new public keys to sets of recipients, and O_{Corrupt} reveals a private key generated by one of the other oracles. The only way for the adversary to win is to perform some sequence of oracles calls and to correctly guess a private key that it should not be authorized to learn.

For any sequence of oracle calls made by the adversary in the game, there is an associated Directed Acyclic Graph (DAG) that represents the legitimate ways to learn the private keys. Each vertex in this DAG corresponds to an entry in the challenge table, and each edge corresponds to

a statement in the derivation table. For example, consider an adversary that makes the following oracle calls:

1. $O_{\text{KeyGen}}()$, generating (pk_1, sk_1)
2. $O_{\text{KeyGen}}()$, generating (pk_2, sk_2)
3. $O_{\text{Encapsulate}}(\{pk_1, pk_2\})$, generating (pk_3, sk_3), (pk_4, pk_4)
4. $O_{\text{KeyGen}}()$, generating (pk_5, sk_5)
5. $O_{\text{Corrupt}}(pk_5)$
6. $O_{\text{Encapsulate}}(\{pk_1\}, \{pk_3, pk_5\})$, generating (pk_6, sk_6), (pk_7, sk_7)
7. $O_{\text{Encapsulate}}(\{pk'\}, \{pk_4\})$, generating (pk_8, sk_8), (pk_9, sk_9)
8. $O_{\text{Encapsulate}}(\{pk_2, pk_4\})$, generating (pk_{10}, sk_{10})
9. $O_{\text{Encapsulate}}(\{pk_1, pk_3\}, \{pk_{10}\})$, generating (pk_{11}, sk_{11}), (pk_{12}, sk_{12})

After these calls, the corruption log contains entries for pk_5, pk_6, pk_7, pk_8, and pk_9. The DAG constructed from the call sequence is visualized in Figure 8.5.

To continue the proof, it is necessary to define the *height* of a challenge table entry in the key secrecy game. The height of the entry produced by a call to O_{KeyGen} is zero. If a challenge table entry has an associated entry in the corruption table, then the height of the entry is undefined. The height of an uncorrupted entry produced by a call to $O_{\text{Encapsulate}}(R_1, \ldots, R_m)$ is $\max(R_{i,j}) + 1$ where the maximum value is taken over all $1 \leq i \leq m$ and $1 \leq j \leq |R_i|$. In the aforementioned example, $\{pk_1, pk_2\}$ have height 0, $\{pk_3, pk_4\}$ have height 1, pk_{10} has height 2, and $\{pk_{11}, pk_{12}\}$ have height 3. Note that the height is defined immediately when $O_{\text{Encapsulate}}$ is called, but a subsequent call to O_{Corrupt} may cause it to become undefined. However, the height can only transition from being defined to being undefined in the case where the height for each $R_{i,j}$ was defined at the time of the $O_{\text{Encapsulate}}(R_1, \ldots, R_m)$ call, but one of these heights became undefined after a subsequent corruption.

The proof proceeds as a sequence of games. **Game 0** is the BRAKEM key secrecy game played against $\text{BRAKEM}_x^{\text{DDL}}$ for some burner public key x. **Game 1** is the same as Game 0 except the intermediate result from Section 8.4.2.2.3 is used to indistinguishably replace all $O_{\text{Encapsulate}}$ calls producing challenge entries of height 1. This is possible because the recipient public keys for an encapsulation of height 1 must all be associated with entries of height 0—in other words, the

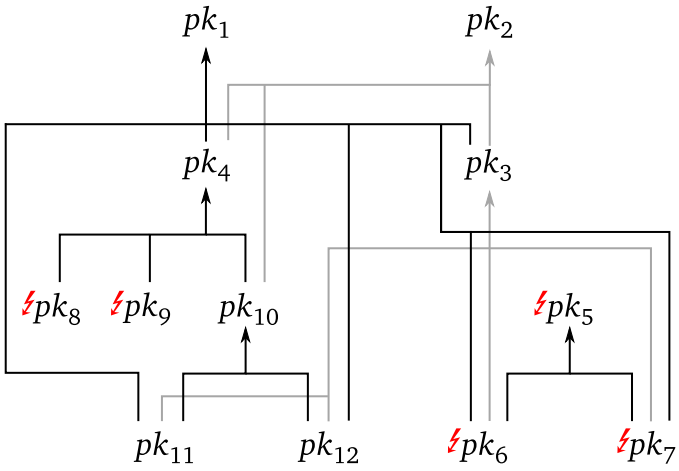


Figure 8.5 A BRAKEM KEY SECRECY DERIVATION GRAPH. The DAG corresponds to the derivation table in a BRAKEM key secrecy game. Each public key pk_i corresponds to an entry (pk_i, sk_i) in the challenge table. Each edge represents an implication in the derivation table. Edge shading is for ease of presentation and has no other meaning. Public keys with a ↯ symbol have an entry in the corruption log. (Ref: 268)

output of calls to O_{KeyGen}. Height 0 entries are fully controllable in a reduction, so the intermediate result applies. Consequently, all entries with height 1 also become fully controllable. The games proceed in this pattern. **Game** i is the same as Game $i-1$ except that all encapsulations for entries with height i are indistinguishably replaced using the intermediate result.

If the adversary wins Game i, then they must have chosen to attack an uncorrupted challenge table entry (pk, sk). In the DAG associated with the oracle calls, this means that the subtree rooted at pk contains no nodes with entries in the corruption log. The key pairs for all of these entries have been generated according to the intermediate result from Section 8.4.2.2.3 by definition of the game, and so any PPT adversary has negligible advantage guessing sk in Game i. The total number of games in this sequence depends on the height of the entry for pk, which depends on the oracle calls made by the adversary. In the worst-case scenario, a PPT adversary can only win in a scenario where the height of the entry to attack is equal to the total number of calls to $O_{\text{Encapsulate}}$, which is polynomially bounded. Consequently, the probability of distinguishing Game 0 from Game i is i times a negligible value, which remains negligible. This completes the proof sketch.

8.5 BRAKEM from zk-SNARKs

Section 8.1.1 presented a general construction for BRAKEM, and Section 8.2 described a construction that is based on special-purpose zero-knowledge proofs of knowledge of double discrete logarithms. However, Schnorr proofs [Sch91] and derivatives are only one method for building NIZKPKs—many other proof systems exist. Zero-knowledge proof systems come in many forms, each with differences in terms of which statements they can prove, performance characteristics, security assumptions, and setup requirements. The general construction for BRAKEM can use one of these alternative proof systems in order to achieve different performance and security properties. This section provides a BRAKEM construction that uses a more recent proof system.

8.5.1 Recent Zero-Knowledge Proof Systems

When choosing a zero-knowledge proof system, the following properties of the proof system are important considerations:

- **Prover time complexity**: the time that it takes to prove a statement with the system. In the context of BRAKEM, this is how long it takes to complete a BRAKEM.Encapsulate call. If it takes a long time to perform encapsulation, then this ultimately restricts the rate at which the Safehouse group state can be updated, thereby limiting the maximum practical group size.
- **Verifier time complexity**: the time that it takes to verify a statement with the system. In the context of BRAKEM, this is how long it takes to complete a BRAKEM.Verify call. Since BRAKEM.Decapsulate uses BRAKEM.Verify as a subroutine, it is also affected by the verifier time complexity. BRAKEM is used to share private keys with many recipients, so there are usually far more calls to BRAKEM.Verify than to BRAKEM.Encapsulate. This means that it is very important to minimize the cost of verifying proof statements.
- **Communication complexity**: the transmission size of proofs. In the context of BRAKEM, this impacts the size of $C_1, \ldots, C_m$, and π. Some zero-knowledge proof systems have an interesting property called *succinctness*. A succinct proof system always generates proof of a constant size with respect to the security parameter, regardless of the size of the proof statement. It is important for the proof sizes to be small for typical BRAKEM statements because many proofs will need to be transmitted during the course of a normal Safehouse conversation.
- **Long-term storage requirements**: the amount of data in long-term storage that is needed to use the proof system. Some zero-knowledge proof systems require large amounts of static

data storage in order to prove or verify statements. It is important to consider the practical implications of these requirements, especially since Safehouse is intended to be used in mobile communication scenarios. Smartphones—particularly low-cost devices—often have very limited non-volatile storage space available for applications that do not come with the operating system, so secure messaging tools that require several gigabytes of data to be permanently stored would likely frustrate users.

- **Trusted setup requirements**: some proof systems require some sort of trusted setup to be performed before they can be used. Some systems merely require the existence of an unstructured common reference string that has been chosen uniformly at random from some set. A trusted random value like this can be used as input to a "nothing up my sleeve" procedure that derives other parameters, such as the primes required to set up the groups for $\text{BRAKEM}^{\text{DDL}}_{\star}$ in Section 8.2.1. Other proof systems require a trusted party to perform a complex procedure that outputs structured data, which may include securely erasing intermediate values (sometimes called "toxic waste"). To somewhat mitigate the weakness of this requirement, it is often possible to use a cryptographic protocol to break up a trusted party into multiple participants in such a way that the trusted setup as a whole remains secure as long as some proportion of the participants behaved honestly.

- **Security assumptions**: the problems that must be difficult in order for the proof system to be secure. A secure proof system is sound (which prevents a prover from proving a statement unless they can compute a valid witness) and zero-knowledge (which prevents a verifier from learning anything about the witness). The generic BRAKEM construction in Section 8.1.1 requires only the DDH problem in the public key group to be difficult, so any additional requirements needed by the proof system (e.g., the additional hardness assumptions in Section 8.4.2.2.1 required by $\text{BRAKEM}^{\text{DDL}}_{\star}$) will expand the assumption set for the final construction.

In recent years, many powerful new zero-knowledge proof systems have emerged. This proliferation has largely been driven by the needs of cryptocurrencies, which often use sophisticated cryptography to differentiate themselves from established competitors and generate lucrative speculation. Motivations notwithstanding, most of these proof systems have solid theoretical foundations, highly optimized implementations, support for a wide range of proof statements, and unprecedented performance characteristics. Particularly interesting proof systems that have recently emerged include: Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs), the most prominent of which are the systems from Gennaro et al. [GGPR13], Groth [Gro16], and their derivatives; Zero-Knowledge Succinct Transparent Arguments of Knowledge (zk-STARKs) [BBHR19]; and Bulletproofs [BBB+18]. These systems all

Table 8.3 A COMPARISON OF ZERO-KNOWLEDGE PROOF SYSTEMS FOR BRAKEM. n refers to the number of gates in the circuit that implements the proof statement. (Refs: 272, 273, and 284)

	zk-SNARKs	zk-STARKs	Bulletproofs
Prover time	$O(n \log(n))$	$O(n \operatorname{polylog}(n))$	$O(n \log(n))$
Verifier time	$O(1)$	$O(\operatorname{polylog}(n))$	$O(n)$
Communication	$O(1)$	$O(\operatorname{polylog}(n))$	$O(\log(n))$
Long-term storage	$O(n)$	$O(1)$	$O(n)$
Trusted setup with waste	Yes	No	No
Security assumptions	Knowledge of exponent, bilinear DH	Hash collision resistance	Discrete logarithm

represent the statement to prove in terms of an arithmetic or boolean circuit, allowing them to prove arbitrary statements from a large class of languages. Table 8.3 provides a brief comparison of these proof systems.

zk-SNARKs are notable due to being *succinct*, so the verification time complexity and communication size for a zk-SNARK are always constant for any statement. They are also more efficient than zk-STARKs in terms of proving time. One weakness of zk-SNARKs is that they require large long-term "proving" and "verifying" master keys to be stored on the device, and these keys are specific to the circuit. Another problem with zk-SNARKs is that they require a trusted setup in which the trusted entity must securely erase "toxic waste" values; an adversary with access to these values can undermine the security of the scheme. The difficulty of handling toxic waste values has caused real-world vulnerabilities: the original zk-SNARK system used by the Zcash cryptocurrency accidentally exposed too many values [Gab19], leading to counterfeiting vulnerabilities in several projects.[32] Finally, the security of zk-SNARKs relies upon stronger hardness assumptions than other schemes. zk-SNARK constructions rely not only on the finite-field and elliptic curve DL problems, but also pairing-based cryptographic assumptions.[33] These pairing assumptions are not yet as mature as DL assumptions in traditional groups; parameter adjustments to mitigate attacks have been required relatively recently [KB16]. A more troubling concern is that zk-SNARKs rely on an extremely powerful assumption called the Knowledge of Exponent Assumption (KEA) [Dam92; HT98; BP04], which states that certain group elements cannot be computed without "knowledge" of a particular exponent. In other words, the KEA states that for

[32] ^ This vulnerability was eliminated in Zcash eight months later by accelerating a planned move to a different zk-SNARK system and deleting the trusted setup transcript under the guise of a false cover story. [SWB19]

[33] ^ zk-SNARKs use variants of the DH assumption in bilinear groups. See the ECRYPT II report [BBC+13, §6] for an overview of pairing assumptions.

every adversary that can compute certain elements, there exists an extractor program that can recover a particular exponent. This assumption is problematic because it is not "falsifiable" in the traditional cryptographic sense: disproving the assumption requires showing that no extractor can possibly exist for a particular adversary [BP04]. It appears that the succinctness property of zk-SNARKs cannot be achieved without unfalsifiable assumptions like the KEA [GW11; BCCT12].

zk-STARKs appear to provide better security than zk-SNARKs. They abandon succinctness, scaling proof size polylogarithmically in terms of the gate count, in exchange for avoiding a trusted setup with toxic waste or storing large per-circuit keys. Moreover, their security relies only on the existence of collision-resistant hashes, which is considered to be a very safe assumption.[34] Unfortunately, zk-STARKs have the greatest proving time complexity of the three systems. Bulletproofs provide yet another point in the security-performance space. Like zk-STARKs, they do not require a trusted setup with toxic waste, and they rely only on the DL problem. Compared to zk-STARKs, they take longer to verify and require more long-term storage, but take less time to prove and produce smaller proofs.

In the context of BRAKEM and its usage within Safehouse, the properties listed in Table 8.3 are the most important considerations. Based on the current state of systems in terms of both academic research and implementation maturity, zk-SNARKs are the most viable alternative to application-specific DDL proofs. They also provide substantially different performance characteristics that may be useful in certain secure group messaging settings. The next section describes how to build a BRAKEM implementation using a zk-SNARK library. However, research into new zero-knowledge proof systems is currently progressing at a very fast pace, and superior implementations are emerging on a yearly basis. Interested readers should consult a living reference document such as `https://zkp.science/` to follow the latest advancements in this field.

8.5.2 Jubjub

The DDL-based construction of BRAKEM presented in Section 8.2 used "double decker" subgroups of finite fields, as described in Section 8.2.1. It was necessary to work within these groups in order to use the application-specific DDL NIZKPKs derived from the original Schnorr protocol [Sch91]; these NIZKPKs require that the group operation is modular multiplication, which allows the proof system to compute values "in the exponent". zk-SNARKs operate in a completely different way and have no such constraints. Consequently, constructions based on zk-SNARKs are free to use more efficient key spaces, such as elliptic curve groups. However, some proof systems are optimized to

[34] This also makes zk-STARKs quantum-resistant, which is one of their major selling points. However, this is irrelevant in the context of BRAKEM since it relies on the DL problem, which is expected to be easy for quantum computers to break with current group parameters.

prove statements about elements from particular groups more efficiently than others. This means that it is important to choose a system that has been architected with the intended application in mind. For a BRAKEM construction, the proof system must be able to efficiently prove that the secret plaintext for a publicly known ElGamal ciphertext corresponds to a publicly known public key. A perfect candidate is Groth's zk-SNARK [Gro16] instantiated with the BLS12-381 curve and used to prove statements about points on the Jubjub twisted Edwards curve [Ele19]. This system was specifically designed to optimize proofs about Pedersen commitments [Ped91] in elliptic curve groups. This optimization means that the zk-SNARK provides very good performance for secret scalar multiplication, which is exactly what is needed for proofs about ElGamal ciphertexts. The prototype implementation of Safehouse includes an implementation of BRAKEM using this proof system as implemented by the `gnark` library [BGP20].

This section uses additive notation for points in elliptic curve groups. For an elliptic curve point P, the x coordinate is denoted $P.x$, and the y coordinate is denoted $P.y$.

zk-SNARKs are very complicated and a full explanation of their construction is outside the scope of this work. Nonetheless, it is important to give a quick overview of the BLS12-381 and Jubjub interaction to give context to the BRAKEM scheme. BLS12-381 [Ele17] is a pairing-friendly elliptic curve constructed according to the BLS formula [BLS03]. Its main curve is defined in the field $\mathbb{F}_q$. The curve equation is:

$$y^2 = x^3 + 4$$

The field modulus is a 381-bit prime:

$$q = \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f624}$$
$$\texttt{1eabfffeb153ffffb9feffffffffaaab}$$

The pairing function is defined in terms of two source groups, G_1 and G_2, and a target group, G_T. All three groups have the same 255-bit prime order:

$$r = \texttt{0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001}$$

G_1 is defined as a subgroup of the curve with the following generator:

$$G_1.x = \texttt{0x17f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac58}$$
$$\texttt{6c55e83ff97a1aeffb3af00adb22c6bb}$$
$$G_1.y = \texttt{0x08b3f481e3aaa0f1a09e30ed741d8ae4fcf5e095d5d00af600db18cb2c04b3ed}$$
$$\texttt{d03cc744a2888ae40caa232946c5e7e1}$$

G_2 is defined as a subgroup of the following curve in $\mathbb{F}_{q^2}$, which is the sextic twist of the curve equation for G_1 in $\mathbb{F}_{q^{12}}$ with basis $\{1, u\}$:

$$y^2 = x^3 + 4(1+u)$$

The generator of G_2 is:

$$\begin{aligned}
G_2.x = {} & \texttt{0x24aa2b2f08f0a91260805272dc51051c6e47ad4fa403b02b4510b647ae3d177} \\
& \texttt{0bac0326a805bbefd48056c8c121bdb8} + \\
& \texttt{0x13e02b6052719f607dacd3a088274f65596bd0d09920b61ab5da61bbdc7f504} \\
& \texttt{9334cf11213945d57e5ac7d055d042b7e} \cdot u \\
G_2.y = {} & \texttt{0xce5d527727d6e118cc9cdc6da2e351aadfd9baa8cbdd3a76d429a695160d12c} \\
& \texttt{923ac9cc3baca289e193548608b82801} + \\
& \texttt{0x606c4a02ea734cc32acd2b02bc28b99cb3e287e85a763af267492ab572e99ab} \\
& \texttt{3f370d275cec1da1aaa9075ff05f79be} \cdot u
\end{aligned}$$

G_T is a finite field group in $\mathbb{F}_{q^{12}}$. The pairing function is defined according to the BLS formula [BLS03]. These parameters are sufficient to instantiate Groth's zk-SNARK [Gro16] and produce proofs about values in $\mathbb{F}_r$.

Jubjub [Ele19] is a twisted Edwards elliptic curve [BBJ+08] with field modulus r, meaning that its point coordinates are values within the "exponent" field of BLS12-381. This means that the zk-SNARK can efficiently prove statements about Jubjub point coordinates. Jubjub is defined by the following curve equation in $\mathbb{F}_r$:

$$-x^2 + y^2 = 1 - \left(\frac{10240}{10241}\right)x^2y^2$$

In the BRAKEM construction, BRAKEM keys and ElGamal ciphertexts all take place in a subgroup on the Jubjub curve. This subgroup has a 252-bit prime order:

$$s = \texttt{0xe7db4ea6533afa906673b0101343b00a6682093ccc81082d0970e5ed6f72cb7}$$

The base point B for the subgroup in Jubjub is given by:

$$\begin{aligned}
B.x &= \texttt{0x33cab9428c97922a456d01156a3c0b5f48bf54819101b9e685ff90bcf4f8ba5e} \\
B.y &= \texttt{0x56f168560e13820f1312a53c8338679b15f8e764499f87033728c334cc269cfb}
\end{aligned}$$

The number of rational points on the curve (i.e., solutions to the curve equation with x and y in $\mathbb{F}_r$) is:

#Jubjub =
0x73eda753299d7d483339d80809a1d8053341049e6640841684b872f6b7b965b8

The cofactor of the subgroup generated by B is 8. In other words, $8s$ = #Jubjub.

8.5.3 ElGamal on Jubjub

To implement BRAKEM, ElGamal must be instantiated using the Jubjub subgroup in order to securely transmit private keys. This is mostly straightforward, except that plaintexts encrypted with ElGamal must be group elements. Moreover, any encoding that is applied to the private keys in order to transform them into group elements must be provable using Groth's zk-SNARK system on BLS12-381. The encoding must also be efficiently invertible, because the original private key must be recovered during decryption; this excludes typical "hash to curve" techniques, which do not need to decode points back into the original values [FSS+20]. Luckily, unlike in the DDL setting (where this problem is handled as described in Section 8.2.4.2), a simple and inexpensive solution is available in the zk-SNARK setting.

Algorithm 8.5 presents a procedure that encodes an input value z as a point (x, y) in the Jubjub subgroup. The encoding takes a parameter ℓ that controls a tradeoff between security of the overall BRAKEM construction and probability of encoding failure. It begins by setting the x coordinate of a test point to $z \cdot 2^\ell$. Next, the curve equation is rearranged to express the y coordinate in terms of the x coordinate:

$$y = y^2 = \frac{-x^2 - 1}{\left(-\frac{10240}{10241}\right)x^2 - 1} \pmod r$$

From the rearranged curve equation, if y happens to be a quadratic residue modulo r, then there are two points (x, y) and $(x, -y)$ on the Jubjub curve with $y = (\pm y)^2 \pmod r$; otherwise, there are no curve points with the given x coordinate. The encoding algorithm tests both potential points to see if either one is part of the Jubjub *subgroup* (not just on the curve). This is done by scalar multiplying the point with the group order, s, and checking to see if the result is the identity element. If the x coordinate does *not* correspond to a subgroup element (either because there are no corresponding points or because neither is in the subgroup), then Algorithm 8.5 increments

Algorithm 8.5 MESSAGE ENCODING FOR ELGAMAL ON JUBJUB. O denotes the identity element. ℓ is an implicitly specified parameter. (Refs: 276 [a] [b], 277 [a] [b], and 279)

Subroutine | PointEncode(z) $\rightarrow (x, y, o)$

Compute $x_0 \leftarrow 2^\ell \cdot z$.
for $(o \leftarrow 0; o < 2^\ell; o{+}{+})$ {
  Compute $x \leftarrow x_0 + o$.
  Compute $y \leftarrow \left(-x^2 - 1\right) \cdot \left(-\frac{10240}{10241} \cdot x^2 - 1\right)^{-1} \pmod r$.
  if (Jacobi symbol of y is 1) {
    Find $y = \sqrt{y} \pmod r$ using Tonelli-Shanks.
    if $(s \cdot (x, y) = O)$ **return** (x, y, o).
    if $(s \cdot (x, -y) = O)$ **return** $(x, -y, o)$
  }
}
return $\perp$.

the x coordinate and tries again. This linear scan continues through the range $[z \cdot 2^\ell, (z+1) \cdot 2^\ell)$ to locate a subgroup element. If no element is found, the algorithm fails.

When the PointEncode algorithm succeeds, the output point will have an x coordinate equal to $z \cdot 2^\ell + o$, where o is an offset in $[0, 2^\ell)$. Decoding a point $P = (x, y)$ back into z is as simple as discarding the lower ℓ bits: $z = \lfloor x/2^\ell \rfloor$. The modular square root that is computed in Algorithm 8.5 can be found using the Tonelli-Shanks algorithm since $r = 1 \pmod 8$.

To prevent the point encoding procedure from failing in practice, it is important to choose a large enough ℓ so that the failure probability is negligible. Each of the r possible x coordinates corresponds to either zero or two curve points, so a total of $\#\text{Jubjub}/2 \approx r/2$ choices will have corresponding points on the curve. Of the points on the curve, $1/8$ of them are in the subgroup. Unfortunately, precisely determining the overall probability that the point encoding procedure succeeds requires knowledge of the distribution of subgroup elements in terms of x coordinates. While several bounds are known (e.g., the results from Fouque et al. [FJT13]), more theoretical work must be done before an exact probability can be derived. In the meantime, it is easy to empirically estimate the failure probability of Algorithm 8.5 by testing many z values drawn uniformly at random and tabulating the resulting offsets (i.e., the return value o). Figure 8.6 shows the distribution of offsets required to encode 80,000,000 values selected randomly in an experiment. These values almost perfectly fit a geometric distribution with probability $p = 1/8$ according to a χ^2 statistical test. The X^2 test statistic for the experimental data was 153 when comparing to the geometric distribution with 1023 degrees of freedom (for $\ell = 10$), so

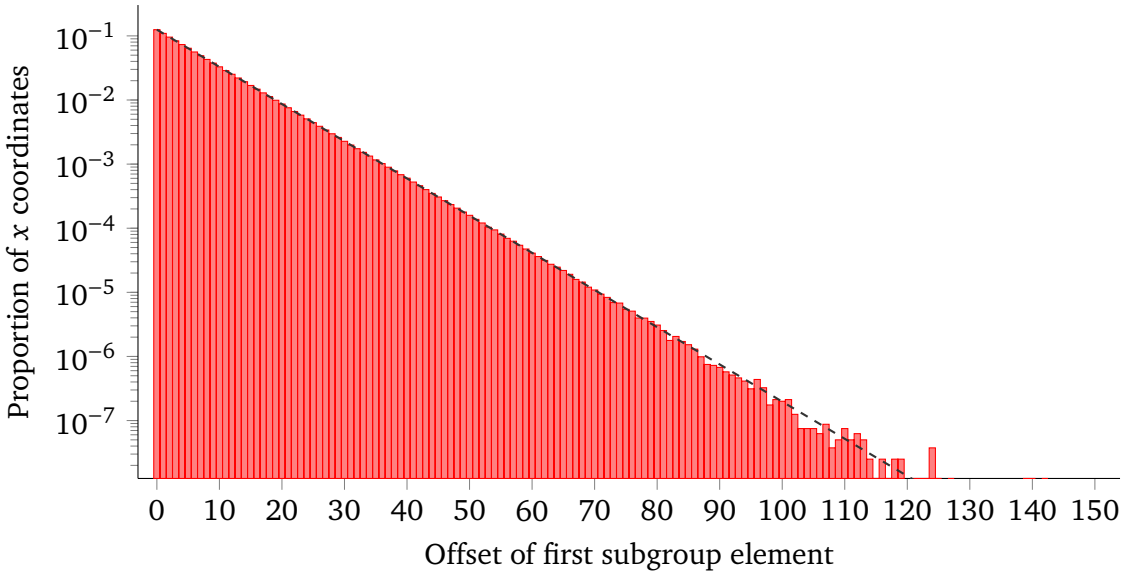


Figure 8.6 DISTRIBUTION OF OFFSETS FOR ENCODING UNIFORMLY RANDOM VALUES AS POINTS. Each bar shows, on a logarithmic scale, the proportion of the 80,000,000 sampled x coordinates with the given offset to the first subgroup element. The results approximate the true probability density function. The dashed line is an interpolation of the geometric distribution for success probability $p = 1/8$. The largest offset observed in the experiment was 142. The logarithmic y axis emphasizes errors in smaller proportions, such as the ones on the right of the graph. (Ref: 277)

Algorithm 8.6 A CRYPTOSYSTEM FOR PRIVATE KEYS USING ELGAMAL ON JUBJUB. The system uses the PointEncode function to encode plaintexts in $[2, t)$. The functions return additional information in order to facilitate the use of zk-SNARKs. (Ref: 279)

Subroutine | EGRand() $\rightarrow (w, C_1)$

Choose $w \xleftarrow{\$} [2, s)$.
Compute $C_1 \leftarrow w \cdot B$.
return (w, C_1).

Subroutine | EGEnc$(RPK, w, sk^*) \rightarrow (C_2, y, o)$

if $(sk^* \notin [2, t))$ { **return** $\perp$ }
Execute $(x, y, o) \leftarrow$ PointEncode(sk^*).
Compute $C_2 \leftarrow w \cdot RPK + (x, y)$.
return (C_2, y, o).

Subroutine | EGDec$(rsk, C_1, C_2) \rightarrow sk^*$

Compute $D = rsk \cdot C_1$.
Compute $P = C_2 - D$.
Compute $sk^* = \lfloor P.x / 2^\ell \rfloor$.
return sk^*.

the distributions match with overwhelming confidence. This means that x coordinates drawn uniformly at random from any small range $[z \cdot 2^\ell, (z+1) \cdot 2^\ell)$ correspond to a subgroup element with essentially a uniform probability of $1/8$. When $\ell = 10$, this suggests that the failure probability of the point encoding function is $(7/8)^{1024} = 2^{-197}$, which can be safely ignored.

Let $t = \lfloor r / 2^\ell \rfloor$. Since PointEncode needs ℓ reserved bits to reduce the probability of its failure condition, the encoded cryptosystem can only encrypt numeric values in the range $[0, t)$. If the goal is to encrypt private keys, then the private keys must be sampled from $[2, t)$ instead of $[2, s)$. This is why greater values of ℓ reduce the discrete logarithm security of keys in the overall BRAKEM construction. Fortunately, since $s < r$, the security level is actually reduced by *fewer* than ℓ bits. When drawing from $[2, t)$ instead of $[2, s)$, private key sizes will use $\log_2(s/t)$ fewer bits. When $\ell = 10$, private keys are approximately 7 bits smaller, leading to a loss of approximately 3.5 bits of security.

Algorithm 8.6 depicts the ElGamal cryptosystem defined on Jubjub and using PointEncode to derive the plaintexts. The interface is defined at a very low level because the overall BRAKEM construction reuses ElGamal randomness C_1 for multiple ciphertexts. EGRand creates a new ElGamal ephemeral key pair with the secret w sampled from the full $[2, s)$ range. A new private key sk^* can then be encrypted to a recipient public key RPK using EGEnc, which employs the point encoding function from Algorithm 8.5. Only the ciphertext (C_1, C_2) needs to be transmitted in order for the recipient to call EGDec and recover sk^*. However, EGEnc returns the low level information y and o so that it can be used by the caller as input to a zk-SNARK.

8.5.4 Verifiable ElGamal NIZKPK: Π_1

The next step in the BRAKEM construction is to introduce a method to publicly verify the correctness of an ElGamal ciphertext. Given a new key pair (pk^*, sk^*) such that $sk^* \in [2, t)$ and $pk^* = sk^* \cdot B$, and a recipient public key RPK, a sender can produce an ElGamal ciphertext by calling $(w, C_1) \leftarrow \text{EGRand}()$ and then $(C_2, y, o) \leftarrow \text{EGEnc}(RPK, w, sk^*)$. A proof must be generated that shows that (C_1, C_2) is a valid encryption of the private key corresponding to pk^* using PointEncode. This proof, denoted Π_1, corresponds to the following statement, given in Camenisch-Stadler notation [CS97]:

$$
\begin{aligned}
&\Pi_1\{(w, sk^*, o, y) : pk^* \sim (C_1, C_2) \sim RPK\} := \\
&PK\{(w, sk^*, o, y) : pk^* = sk^* \cdot B \wedge \\
&\qquad\qquad C_1 = w \cdot B \wedge \\
&\qquad\qquad 0 \leq o < 2^\ell \wedge \\
&\qquad\qquad (sk^* \cdot 2^\ell + o, y) \text{ is on the curve} \wedge \\
&\qquad\qquad C_2 = w \cdot RPK + (sk^* \cdot 2^\ell + o, y)\ \}
\end{aligned}
$$

It is not strictly necessary for Π_1 to verify that the ElGamal plaintext is in the correct subgroup—verifying that it is on the curve is sufficient. If the plaintext is maliciously on the curve but outside the subgroup, the recipient will still decrypt sk^* correctly. Although such a ciphertext would leak information about sk^*, a malicious prover is free to simply send sk^* to whomever it wishes in any case.

Efficiently implementing Π_1 requires a very powerful proof system, such as a zk-SNARK. Implementations of Groth's proof system [Gro16] like the `gnark` library typically allow the developer to specify the proof statement in terms of an arithmetic circuit with a mix of public (known to both the prover and verifier) and private (known to the prover only) inputs. Values in this circuit come from $\mathbb{F}_r$, which is the field in which Jubjub is defined. The developer can then add arithmetic constraints in the circuit that must be satisfied by the prover's witness. Some of these constraints may yield intermediate values that can then be used as inputs to further constraints.

To implement Π_1, the following constraints must be encoded in the circuit, where all variables are elements in $\mathbb{F}_r$:

- $a = b$.
- $a \leq b$.

- $a = -b \pmod{r}$.
- $a = b + c \pmod{r}$.
- $a = b \cdot c \pmod{r}$.
- $a \in \{0, 1\}$.
- If $a = 0$ then $b = c$. If $a = 1$ then $b = d$.
- $b_1, \ldots, b_m$ are the m-bit binary representation of a, where m is public.
- (x, y) is a point on the Jubjub curve.
- $(x', y') + (x'', y'') = (x, y)$.
- $a \cdot B = (x, y)$, where B is the Jubjub base point.
- $a \cdot (x', y') = (x, y)$.

Some of these constraints can be constructed using the earlier ones. Testing to see if a point is on the curve can be done using the curve equation (see Section 8.5.2) and the basic modular arithmetic constraints. The final three constraints involve point addition. The fact that Jubjub is a twisted Edwards curve means that a *complete* point addition formula is available [BBJ+08, §6]. This means that point addition can be implemented using the basic arithmetic constraints without any special cases. Scalar multiplication by fixed or variable bases can be implemented from the point addition circuit using the standard "double-and-add" algorithm: the scalar is first expanded into its binary form, and then the conditional selection circuit is used to determine when to perform point addition based on the corresponding bit.[35] The `gnark` library implements all of the required circuits off-the-shelf. With this library of circuits, all of the required constraints in Π_1 can be specified and proven by the zk-SNARK system.

8.5.5 A BRAKEM Construction using zk-SNARKs

It is now possible to define a complete BRAKEM construction given the ElGamal cryptosystem in Section 8.5.3 and the Π_1 NIZKPK in Section 8.5.4. The overall approach is simply the general construction from Section 8.1.1: for each ring of receivers, encrypt a new private key to each of them using EGEnc, reusing a single call to EGRand for the whole ring to improve performance.

[35] ^ This is effectively Algorithm 8.3 written in additive notation and performed within the zk-SNARK.

For each ring, the Encapsulate function uses Π_1 to produce a proof that the ciphertext sent to the first receiver is valid and corresponds to the new public key. A secondary proof shows that all ciphertexts for the ring encrypt the same plaintext.

To show that multiple ElGamal ciphertexts in the Jubjub subgroup encrypt the same plaintext, it is possible to use another zk-SNARK. However, it makes more sense to use a Schnorr proof [Sch91] for this NIZKPK. Although the proof size is larger than a zk-SNARK, the size advantage is minimal compared to the significantly improved proving time (unlike for Π_1, whose only alternative is a very large DDL proof). If the ring only has two elements, then this is essentially a DLEQ proof (see Section 8.2.2.2) adapted to an elliptic curve setting. When the ring has more than two elements, then the proof is a BDLEQ proof (see Section 8.2.2.3) adapted to an elliptic curve setting. Let Elliptic Curve Plaintext Equality (EC-PTEQ) denote this conditional proof using the same input and output parameters as BDLEQ, but simply performing a DLEQ proof when the ring has only two recipients. Note that DLEQ and BDLEQ were previously defined using exponential notation in finite fields, but they operate equivalently in elliptic curve groups using additive notation.

The resulting BRAKEM construction is called BRAKEM^{ZK}. It operates as follows:

- **BRAKEM^{ZK}.KeyGen**$(s') \rightarrow (pk, sk)$:

  Choose sk in $[2, t)$ using randomness in s'.
  Compute $pk \leftarrow sk \cdot B$.

- **BRAKEM^{ZK}.Encapsulate**$(R_1, \ldots, R_m; s') \rightarrow (pk_1^*, \ldots, pk_m^*, sk_1^*, \ldots, sk_m^*, C_1, \ldots, C_m, \pi)$:

  Derive all subsequently required randomness from s'.
  for each $(1 \leq i \leq m)$ {
  Generate key pair $(pk_i^*, sk_i^*) \leftarrow$ BRAKEM^{ZK}.KeyGen().
  Execute $(w_i, C_{1,i}) \leftarrow$ EGRand().
  for each $(1 \leq j \leq |R_i|)$ {
  Execute $(C_{2,i,j}, y_{i,j}, o_{i,j}) \leftarrow$ EGEnc$(R_{i,j}, w_i, sk^*)$.
  }
  Prove $\pi_{1,i} = \Pi_1\{(w_i, sk_i^*, o_{i,1}, y_{i,1}) : pk_i^* \sim (C_{1,i}, C_{2,i,1}) \sim R_{i,1}\}$.
  Prove $\pi_{2,i} =$ EC-PTEQ$\{w_i : (B, C_{1,i}) \approx$
  $\hookrightarrow \quad (R_{i,2} - R_{i,1}, C_{2,i,2} - C_{2,i,1}) \approx \cdots \approx (R_{i,|R_i|} - R_{i,1}, C_{2,i,|R_i|} - C_{2,i,1})\}$.
  Assign $C_i = (C_{1,i}, C_{2,i,1}, \ldots, C_{2,i,|R_i|})$.
  }
  Assign $\pi = (\pi_{1,1}, \pi_{2,1}, \ldots, \pi_{1,m}, \pi_{2,m})$.

- **BRAKEM^{ZK}.Decapsulate**$(i, j, sk, R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow sk_i^*$:

> **if** (**!** $((1 \leq i \leq m)$**&&**$(1 \leq j \leq |R_i|)))$ **return** $\perp$.
> Execute $v \leftarrow$ BRAKEM^{ZK}.Verify$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi)$
> $\hookrightarrow$ and record the interpretation of C_i.
> **if** $(v = 0)$ **return** $\perp$.
> Compute $sk_i^* \leftarrow \text{EGDec}(sk, C_{1,i}, C_{2,i,j})$.
> **return** sk_i^*.

- **BRAKEM^{ZK}.Verify**$(R_1, \ldots, R_m, pk_1^*, \ldots, pk_m^*, C_1, \ldots, C_m, \pi) \rightarrow b$:

> Interpret C_i for $1 \leq i \leq m$ and π as defined in BRAKEM^{ZK}.Encapsulate.
> **if** (interpretation of any C_i or π fails) **return** 0.
> **for each** $(1 \leq i \leq m)$ {
> **if** ($\pi_{1,i}$ fails to verify) **return** 0.
> **if** ($\pi_{2,i}$ fails to verify) **return** 0.
> }
> **return** 1.

Since zk-SNARKs are succinct, the size of π can be further reduced by coalescing multiple Π_1 statements into a single circuit. This is similar to the core idea of Π_0 (see Section 8.2.7), but without any special batching techniques. This also reduces the verification time to $O(1)$ instead of $O(m)$, since all zk-SNARKs using Groth's system take the same amount of time to verify. The major downside of this approach is that a new proving and verifying key (output by a trusted setup) must be stored for each possible value of m. In practice, this storage requirement grows quadratically and quickly becomes problematic, especially for storage-constrained mobile devices. The prototype implementation caches circuits for values of m between 1 and 5, requiring 72 MiB of storage space. When $m > 5$, the rings are greedily partitioned into chunks of maximum size, with an appropriately sized zk-SNARK for each chunk.

8.6 Choosing a BRAKEM Construction

A Safehouse implementation requires a BRAKEM implementation. This chapter presented three main constructions: BRAKEM$^{2DDL}_{\star}$ (see Section 8.2.4.1), BRAKEM$^{DDL}_{\star}$ (see Section 8.2.8), and BRAKEM^{ZK} (see Section 8.5.5). The main considerations for choosing a BRAKEM construction are security and performance.

$\text{BRAKEM}_\star^{\text{2DDL}}$ is far less efficient than the other two options. It is always preferable to use $\text{BRAKEM}_\star^{\text{DDL}}$ instead of $\text{BRAKEM}_\star^{\text{2DDL}}$ with the possible exception of extremely high security applications in which performance is not a consideration. The main reason to use $\text{BRAKEM}_\star^{\text{2DDL}}$ would be if $\text{BRAKEM}_\star^{\text{DDL}}$ is shown to be insecure for efficient parameter sizes, which would likely require a break of either the discrete logarithm hidden subgroup or multiplicative subgroup rounding with verifiers assumptions (see Section 8.4.2).

Choosing between $\text{BRAKEM}_\star^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$ is more difficult. These two constructions have very different security assumptions. While the security assumptions for $\text{BRAKEM}_\star^{\text{DDL}}$ seem to be inherently safer than the ones for $\text{BRAKEM}^{\text{ZK}}$, such as the unfalsifiable KEA, the zk-SNARK setup used by $\text{BRAKEM}^{\text{ZK}}$ is also used by cryptocurrencies that would lose large amounts of financial securities if the proof system was broken. In theory, this should bring increased expert scrutiny upon the $\text{BRAKEM}^{\text{ZK}}$ assumptions. In any case, the $\text{BRAKEM}^{\text{ZK}}$ proof system requires a trusted setup with "toxic waste" in order to instantiate the system, which is more difficult to accomplish than the setup for $\text{BRAKEM}_\star^{\text{DDL}}$. This can be an important consideration for some deployments.

The asymptotic performance of various proof systems was compared in Table 8.3. However, the DDL-based proof system used by $\text{BRAKEM}_\star^{\text{DDL}}$ is not directly comparable because it is application-specific and does not express the proof statement in terms of gates. The best way to compare the performance of these systems is to perform empirical measurements with real implementations. Table 8.4 compares the performance of the prototype $\text{BRAKEM}_\star^{\text{DDL}}$ implementation described in Section 8.3 with the prototype $\text{BRAKEM}^{\text{ZK}}$ implementation described in Section 8.5. The evaluation includes various dimensionalities for the recipient public key sets input to BRAKEM.Encapsulate.

Several differences between the two schemes are readily apparent. Encapsulation with $\text{BRAKEM}_\star^{\text{DDL}}$ is at least 20 times faster than $\text{BRAKEM}^{\text{ZK}}$ for the same input parameters. This is to be expected, since slow proving times are the primary weakness of current zk-SNARK constructions. In contrast, verification with $\text{BRAKEM}^{\text{ZK}}$ is roughly five times faster than $\text{BRAKEM}_\star^{\text{DDL}}$ in the 1×1 scenario (i.e., a single encapsulation to a single recipient). This also makes sense, since the large modular exponentiations in $\mathbb{G}_1$ required by $\text{BRAKEM}_\star^{\text{DDL}}$ are far more expensive than the zk-SNARK verification function. It is particularly interesting to note the scalability of the verification function for the two schemes. $\text{BRAKEM}_\star^{\text{DDL}}$.Verify takes almost the same amount of time for the 1×1 and 1×10 (i.e., a single encapsulation to 10 recipients) scenarios, which shows that the cost of verifying Π_0 dominates the cost of verifying the BDLEQ proof. As the number of encapsulations increases, $\text{BRAKEM}_\star^{\text{DDL}}$.Verify scales linearly, but with a slope less than one. This shows that the batching technique used in Π_0, which dominates the verification time, is more effective than performing multiple proofs. In contrast, $\text{BRAKEM}^{\text{ZK}}$.Verify takes significantly longer in the 1×10 scenario compared to the 1×1 scenario, showing that the cost of verifying

Table 8.4 A PERFORMANCE COMPARISON OF BRAKEM$^{\text{DDL}}_{\star}$ AND BRAKEM$^{\text{ZK}}$. (Ref: 284)

		BRAKEM$^{\text{DDL}}_{\star}$		BRAKEM$^{\text{ZK}}$	
Encapsulation time [ms]	1×1	4.8	(0.2)	190	(2)
	1×10	7.0	(0.4)	195	(4)
	2×10	13.2	(0.5)	346	(6)
	5×10	32	(1)	767	(10)
Verification time [ms]	1×1	12.5	(0.4)	2.5	(0.2)
	1×10	13.1	(0.4)	5.5	(0.3)
	2×10	18	(2)	7.8	(0.7)
	5×10	33	(3)	13.1	(0.7)
Decapsulation time [ms]	1×1	13.4	(0.5)	2.5	(0.2)
	1×10	13.9	(0.5)	5.6	(0.2)
	2×10	19	(2)	7.9	(0.6)
	5×10	38	(3)	13.7	(0.5)
Communication [B]	1×1	16 449		322	
	1×10	19 905		642	
	2×10	35 682		1 058	
	5×10	83 013		2 306	
Long-term storage [B]	Raw	0		75 727 164	
	7-Zip	0		40 210 973	

The exponentiation tables for BRAKEM$^{\text{DDL}}_{\star}$ were precomputed on startup. zk-SNARK circuits for up to 5 recipients were established for BRAKEM$^{\text{ZK}}$. Standard deviations are in parentheses. $n \times m$ denotes that BRAKEM.Encapsulate was called with n sets of receivers, each containing m receiver public keys.

the EC-PTEQ proof dominates the cost of verifying Π_1. This also makes sense, since zk-SNARKs are much faster to verify than Schnorr proofs. This was a tradeoff made by BRAKEM$^{\text{ZK}}$ to avoid increasing the cost of BRAKEM$^{\text{ZK}}$.Encapsulate even further by using a zk-SNARK for the plaintext equality. This also explains why BRAKEM$^{\text{ZK}}$.Verify scales linearly with a slope that is much closer to one: since a single zk-SNARK verification is needed in all scenarios and the proof system is succinct, the increased cost of verification is solely due to more EC-PTEQ proofs. Decapsulation is marginally slower than verification for both schemes, since it is essentially a verification followed by an inexpensive ElGamal decryption in a small group.

The communication cost also varies substantially between the two schemes. BRAKEM$^{\text{DDL}}_{\star}$ requires roughly 16 KiB of data for each Π_0, and this cost increases linearly with the number of recipient rings. This is due to the low soundness of the cut-and-choose protocol, requiring 128 repetitions of the underlying NIZKPK to compensate. In terms of the proof components described in Section 8.2.7, the $s_{i,j}$ values require 4096 bytes, the $S_{i,j}$ values require 6272 bytes, and the w_i values require 4096 bytes. The remainder of the size comes from the other components of BRAKEM$^{\text{DDL}}_{\star}$.Encapsulate, such as the ElGamal ciphertexts themselves. In contrast, BRAKEM$^{\text{ZK}}$ uses roughly 0.5 KiB for each recipient ring. This space is almost entirely used for the ElGamal ciphertexts, since the zk-SNARK always uses a small constant amount of space. The EC-PTEQ proof is also constant size in terms of the number of recipients in a ring, but one such proof must be included for each ring in the encapsulation.

Finally, no additional storage is required on the device for BRAKEM$^{\text{DDL}}_{x}$. Additional storage may optionally be used to cache exponentiation tables for the burner key x, but this is not required—a client can also generate these tables in memory dynamically. Moreover, it is often faster for modern hardware to recompute the exponentiation tables from scratch instead of loading them from a storage device. In the event that additional storage is used for the exponentiation tables, it is up to the developer to decide how much storage to allocate. In contrast, BRAKEM$^{\text{ZK}}$ requires storage for the proving and verifying keys produced by the zk-SNARK trusted setup. In order for the prototype to store keys for circuits up to five recipients—which is what facilitates the good scalability seen in the results—the device required roughly 72 MiB of storage for the raw data. Luckily, this limitation can be mitigated by using data compression: the 7-Zip utility was able to compress the data to roughly 38 MiB using the LZMA2 algorithm. This shows that larger circuit sizes might be feasible to store on mobile devices, but note that the raw size scales quadratically with the maximum ring count.

Given these results, the choice of BRAKEM implementation requires careful consideration of the target deployment. Assuming that a developer is comfortable with the security assumptions of both schemes, the decision becomes primarily driven by performance. A typical Safehouse group update requires many BRAKEM encapsulations. In large groups, multiple encapsulations in 5×10 or 3×64 recipient ring scenarios are typical, depending on the protocol configuration.

For these situations, the large encapsulation time cost for BRAKEM^{ZK} may become prohibitively expensive, especially for mobile devices. All participants in a Safehouse conversation will need to perform these operations periodically (e.g., on a weekly basis). On the other hand, participants will need to verify BRAKEM proofs constantly. BRAKEM^{ZK} excels at this task. In combination with its small message sizes, this may make BRAKEM^{ZK} more viable for very large groups for which members verify far more NIZKPKs than they produce. Ultimately, the BRAKEM implementations will need to be evaluated in the context of a deployment scenario to determine if any of these performance factors become untenable. The topic of selecting a BRAKEM implementation is revisited in Section 10.13, which presents the results of a performance evaluation in which several Safehouse configurations were used to simulate realistic conversation transcripts.

8.7 Chapter Summary

This chapter introduced BRAKEM, a new cryptographic primitive. This primitive is similar to a KEM, except that multiple secrets are transmitted to multiple sets of recipients. Moreover, these secrets are the private keys corresponding to public keys that can be used as targets for subsequent BRAKEM operations (i.e., the scheme is *recursive*), and the correctness of the encapsulation is publicly verifiable. This chapter presented multiple BRAKEM constructions: constructions based on application-specific DDL NIZKPKs, and an alternative construction based on zk-SNARKs. This chapter also described a prototype implementation along with detailed performance optimization advice, and proved the security of the DDL-based constructions.

Section 8.1 defined the BRAKEM scheme. Section 8.2 introduced a construction based on DDL NIZKPKs. This approach is ultimately based on Schnorr proofs [Sch91]; it provides strong security and very efficient proving and verifying times (dozens of milliseconds), at the cost of relatively large ciphertexts (approximately 20 KiB of data to send a private key to 10 recipients). Section 8.2.2 covered previously known Schnorr-based NIZKPKs that are useful for the construction. Section 8.2.3 presented a new cut-and-choose variant of a cross-group DLEQ NIZKPK that can be efficiently integrated into the rest of the design. Section 8.2.4.1 introduced a very conservative construction of BRAKEM relying on only standard DL security assumptions. Section 8.2.4.2 proposed a far more efficient construction of BRAKEM that relies on unusual but reasonable security assumptions: the *discrete logarithm hidden subgroup* and *multiplicative subgroup rounding with verifiers* assumptions—these were defined and discussed in Section 8.4.2.2.1. Section 8.2.5 combined previously introduced NIZKPKs into useful but inefficient DDL-based NIZKPK that can be used to build verifiable encryption. Section 8.2.6 introduced a novel proof technique reminiscent of the Unruh transform [Unr15] that greatly accelerates the performance of DDL NIZKPKs. Section 8.2.7 defined a large NIZKPK that unifies all of the proof statements required

to implement BRAKEM from DDL assumptions. Section 8.2.8 defined the BRAKEM construction that is actually expected to be used in practice, $\text{BRAKEM}_{\star}^{\text{DDL}}$: a DDL-based construction using the unified NIZKPK and unusual security assumptions.

Section 8.3 described a prototype implementation of $\text{BRAKEM}_{\star}^{\text{DDL}}$ and all of the details required to actually use the construction. Section 8.3.1 covered how to generate and verify group parameters for the scheme—a non-trivial topic that is often overlooked when describing cryptographic constructions. Section 8.3.1.1 outlined attacks against "finite field" DH groups, and how to generate and verify group parameters to avoid or mitigate the attacks. Section 8.3.1.2 presented algorithms for generating large primes with the required structure to implement $\text{BRAKEM}_{\star}^{\text{DDL}}$, which can be prohibitively expensive without the proper optimizations. Section 8.3.1.3 discussed security considerations surrounding the "burner key" technique that is used in $\text{BRAKEM}_{\star}^{\text{DDL}}$ to greatly accelerate encapsulation and decapsulation. Section 8.3.1.4 offered sample group parameters to instantiate $\text{BRAKEM}_{\star}^{\text{DDL}}$. Section 8.3.2 provided detailed coverage of the performance optimizations included in the prototype implementation. Primarily, this coverage consisted of previously known techniques to accelerate modular exponentiation, along with empirically optimized parameters for the $\text{BRAKEM}_{\star}^{\text{DDL}}$ construction and its intended use. Section 8.3.2.2 described a fast modular squaring technique for Intel processors with greater clarity than can be found in Intel white papers. Section 8.3.2.4 provided a highly optimized addition chain for fixed-exponent modular exponentiation with the aforementioned sample parameters. Section 8.3.2.7 pointed out the locations in $\text{BRAKEM}_{\star}^{\text{DDL}}$ for which parallel computation is the most beneficial.

Section 8.4 provided detailed security sketches for the DDL-based BRAKEM construction. Section 8.4.1 included a security proof for the unified NIZKPK. The soundness proof makes use of the novel NIZKPK technique to ensure that certain responses are identical after the extractor rewinds the prover; the use of this assumption appears in Section 8.4.1.3. Section 8.4.2 sketched the security of $\text{BRAKEM}_{\star}^{\text{DDL}}$ as a whole. Notably, Section 8.4.2.2 sketched the key secrecy proof as a nested sequence of games using the new hardness assumptions.

Section 8.5 introduced a completely separate construction for BRAKEM, $\text{BRAKEM}^{\text{ZK}}$, derived from zk-SNARKs. Section 8.5.1 compared recently introduced proof systems that could be used to construct BRAKEM, and explained the rationale for selecting zk-SNARKs. Section 8.5.4 described how to implement the ElGamal cryptosystem [ElG85] on the Jubjub Edwards curve [Ele19]. This curve can be instantiated within BLS12-381 [Ele17] for efficient integration with Groth's zk-SNARK scheme [Gro16]. Section 8.5.4 described the constraints that must be checked in a zk-SNARK circuit to implement BRAKEM. Section 8.5.5 presented the actual zk-SNARK-based BRAKEM construction. This construction has very small ciphertexts (approximately 0.6 KiB of data to send a private key to 10 recipients) and very fast verification times (a few dozen milliseconds or fewer), but it comes with very expensive proving times (hundreds of milliseconds).

The prototype developed as part of this work includes implementations of both $\text{BRAKEM}_{\star}^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$ in the Go programming language. Section 8.6 provided a performance comparison between the two implementations and discussed how to choose between them for a deployment.

The BRAKEM primitive forms the core of the Safehouse secure group messaging protocol. The next chapter introduces the final prerequisite: a new interactive group key exchange protocol that can accelerate mass joining operations, allowing many participants to efficiently join secure group conversations simultaneously.

CHAPTER 9 | TKLL: A New Interactive Group Key Exchange

In this chapter: Protocol

Modern secure group messaging protocols are often implemented using CGKA protocols, as discussed in Chapter 4. In particular, protocols like MLS arrange their key graphs in tree structures, where private keys corresponding to nodes are known by the participants with corresponding leaf nodes in the subtree. Safehouse uses a key tree for similar purposes, even though Safehouse is a conversation security protocol that does not strictly fit the interface of a CGKA. Chapter 3 pointed out that supporting a "mass join" operation, which allows multiple participants to efficiently join the group simultaneously, can help a secure group messaging protocol to support larger groups. This chapter introduces a new interactive GKE protocol called TKLL that can be used to implement the "mass join" operation for group messaging protocols with tree-based states. TKLL can be immediately applied to the design of MLS and Safehouse, but it is also a contribution of independent interest.

As discussed in Chapter 7, in order to join a Safehouse group, members must explicitly consent by interacting with the protocol; it is not possible to unilaterally add someone to a group without their involvement. A member joins the group by receiving an invitation containing secret key material, performing some cryptographic operations using this secret key material, and then sending the output of these cryptographic operations to the server for subsequent distribution to other group members.[1] It is safe to assume that a member is online during the time that they perform these cryptographic operations and deliver the results to the server.[2] Therefore, when several members attempt to join the group at nearly the same time, it is possible for these members to perform an interactive protocol in order to efficiently complete a "mass join" operation. TKLL can be used to prepare a "mass join" message that is then non-interactively sent to the existing members for later processing. The interactive protocol can be subject to relatively

[1] ^ Chapter 10 describes the Safehouse protocol. It covers the details concerning this secret key material, the cryptographic operations, the delivery mechanism, and how other members process the output.

[2] ^ However, there may be a long delay between the time that the party receives the invitation and the time that they decide to use the invitation to join the group by performing the cryptographic operations.

short deadlines; if one of the participants in the interactive protocol goes offline or misbehaves, the protocol can simply restart without the problematic participant. In practice, a server can efficiently aggregate requests to join the group: when new members have queued to join the group, a new TKLL session is started as soon as the previous one has concluded and produced a "mass join" message.

TKLL is an interactive protocol between multiple participants. The protocol takes a key tree as input; this tree describes what key pairs should be established and which participants should learn each private key. The new participants then execute the TKLL protocol interactively. Once complete, each participant receives as output a set of private keys determined by their location in the key tree. TKLL also outputs a ciphertext that can be non-interactively sent to the existing participants in the group. This ciphertext can be processed to verify the correctness of the key tree, effectively "importing" it into the tree-based group state. The overall protocol (which may be a CGKA or a part of Safehouse) can then "merge" the new tree into the existing group state in a protocol-specific manner.

The key insight of the TKLL protocol is that a key tree can be established efficiently using a derivative of the BD GKE, as discussed in Section 3.4. Among the GKE families, the BD protocol family can interactively establish a group key using a minimum number of rounds, which is the most important performance metric for protocols performed over the Internet, where round-trip times are high. TKLL essentially performs a custom variant of the BD protocol for each tree node simultaneously, quickly establishing shared keys for every node in the tree. The key pairs generated by TKLL ultimately need to come from key spaces that are suitable for DH exchanges as part of the overall secure messaging protocol. In the case of Safehouse, the key pairs must be usable as BRAKEM targets (see Chapter 8). Luckily, the BD GKEs involved in TKLL can be performed independently of the target cryptosystem. The shared secret keys produced by the GKEs can input into a KDF to derive a corresponding private key for the target cryptosystem. In practice, the TKLL construction described in this chapter uses elliptic curve groups for the key exchanges. This chapter uses additive notation for elliptic curve groups: B denotes the group generator, sP denotes scalar multiplication of point P by scalar s, and $P + Q$ denotes the elliptic curve group operation (point addition) with points P and Q. q_E denotes the order of B.

Section 9.1 defines the key tree notation used by the remainder of the chapter. Section 9.2 introduces a new GKE in the BD family that is used as a sub-protocol by TKLL; it implements the required properties as simply as possible. Section 9.3 describes an unauthenticated version of TKLL. Section 9.4 extends the previous result to introduce the complete TKLL protocol, including the desired authentication properties.

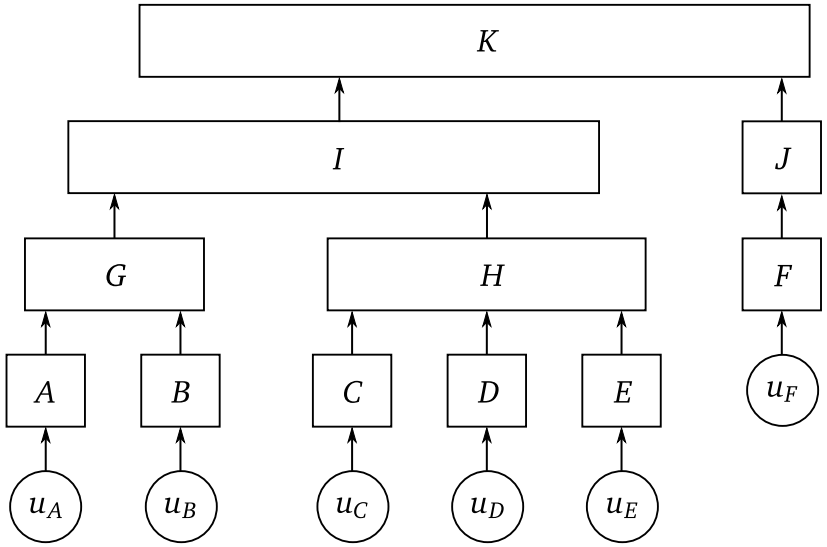


Figure 9.1 AN EXAMPLE OF A KEY TREE. The rectangles are k-nodes and the circles are u-nodes. (Refs: 292 and 293 [a] [b] [c])

9.1 Key Tree Notation

Key trees and their history were presented in Section 3.5. Wong et al. [WGL00] originally proposed general notation for symmetric key graphs; this chapter adopts their notion of k-nodes and u-nodes in the context of asymmetric key trees. Key trees are visualized in this chapter using the same conventions as in Chapter 4.

Figure 9.1 depicts an example of a generic key tree for six participants, each with an associated u-node (labeled u_A through u_F). Each k-node (labeled A through K) is associated with a key pair. All of the public keys for k-nodes are publicly known as part of the key tree data structure. A participant knows the private key associated with a k-node if there is a path from their u-node to that k-node (equivalently, if their u-node is in the subtree rooted at the k-node). The *user set* associated with a k-node v contains all of the participants with knowledge of the private key (i.e., the set of participants with u-nodes in the subtree rooted at v). The user set of v in key tree T is denoted by $\text{userset}_T(v)$. As a natural extension of this notation, $\text{userset}(T)$ denotes all of the participants with u-nodes in the key tree T. Similarly, the *key set* associated with a participant U_x contains all of the k-nodes with private keys known to U_x (i.e., for every k-node that can be reached from the u-node u_x). The key set for a participant U_x in key tree T is denoted by $\text{keyset}_T(U_x)$. A valid key tree T must not have $\text{userset}_T(v) = \emptyset$ for any k-node v in T (i.e., it

must not contain any k-nodes for which no member knows the private key). It is not required for k-nodes in a valid key tree to have unique user sets. For example, the two distinct k-nodes labeled F and J in Figure 9.1 have the same user set $\{u_F\}$. While keys with identical user sets may appear to be redundant, they can be useful for algorithms that use the k-nodes for different purposes after the TKLL protocol has completed.

Since the key trees given to TKLL are intended to be used in a secure group messaging protocol in which any subset of participants may leave the group, all key trees in this chapter are assumed to have at least one k-node known exclusively to any given participant—that k-node is called the participant's *personal key*. In terms of the key tree, this means that every u-node is connected to exactly one k-node that has no other incoming edges. This property can be seen in the example shown in Figure 9.1—the personal keys are A, B, C, D, E, and F.

This chapter also makes use of standard tree definitions in the context of key trees. The unique *parent* node of a node v in key tree T is denoted by $\text{parent}_T(v)$. The parent of the root node is denoted by the special symbol $\top$. The parent of a node is always either a k-node or $\top$ (i.e., u-nodes are always leaves). The set of *children* of a node v in key tree T are nodes in a set denoted by $\text{children}_T(v)$. In general, this set may contain both k-nodes and u-nodes. For any u-node v in key tree T, $\text{children}_T(v) = \emptyset$. Note that for any $v' \in \text{children}_T(v)$, $\text{parent}_T(v') = v$. The set of *sibling* nodes of a node v in key tree T, including v itself, is denoted by $\text{sibling}_T(v)$. Note that $\text{sibling}_T(v) = \text{children}_T(\text{parent}_T(v))$ if v is not the root node, and that $\text{sibling}_T(v) = \{v\}$ if v is the root node. The *depth* of a node v in key tree T is an integer denoted by $\text{depth}_T(v)$. The depth of the root node is zero. For each non-root node v, $\text{depth}_T(v) = \text{depth}_T(\text{parent}_T(v)) + 1$. The *height* of a key tree T, denoted by $\text{height}(T)$, is the height of the tree formed by the k-nodes. Equivalently, $\text{height}(T)$ is the maximum value of $\text{depth}_T(v)$ for any u-node v in T.[3] For example, a key tree with a single k-node has a height of one.

Using the key tree T depicted in Figure 9.1 as an example:

- $\text{userset}(T) = \{U_A, U_B, U_C, U_D, U_E\}$. Note that these refer to the labels for participants and not to k-nodes or u-nodes.
- $\text{userset}_T(G) = \{U_A, U_B\}$ and $\text{userset}_T(I) = \{U_A, U_B, U_C, U_D, U_E\}$.
- $\text{keyset}_T(U_F) = \{F, J, K\}$ and $\text{keyset}_T(U_D) = \{D, H, I, K\}$
- $\text{parent}_T(u_A) = A$, $\text{parent}_T(H) = I$, and $\text{parent}_T(K) = \top$.
- $\text{children}_T(F) = \{u_F\}$, $\text{children}_T(H) = \{C, D, E\}$, and $\text{children}_T(u_E) = \emptyset$.

[3] The deepest node in a valid key tree is always a u-node, since empty user sets are not permitted.

- $\text{sibling}_T(D) = \{C, D, E\}$, $\text{sibling}_T(K) = \{K\}$, and $\text{sibling}_T(u_A) = \{u_A\}$.
- $\text{depth}_T(K) = 0$, $\text{depth}_T(G) = 2$, and $\text{depth}_T(u_E) = 4$.
- $\text{height}(T) = 4$.

9.2 Simplified Kim-Lee-Lee

The TKLL protocol involves performing an interactive GKE for each k-node in the given key tree T. For a given k-node v, the GKE takes place between participants with one representative for each node in $\text{children}_T(v)$. This section describes a simple GKE from the BD protocol family called Simple KLL that can perform this task. The full KLL protocol introduced by Kim et al. [KLL04] and described in Section 3.4 provides functionality that is not required in this context. Simple KLL eschews the dynamism, authentication, and keying byte mechanisms. Essentially, Simple KLL is the original BD protocol [BD94] using the XOR operator (denoted by $\oplus$) from KLL. One practical benefit of this approach is that mature cryptographic libraries designed to perform DH key exchanges can be used to compute intermediate values without modification. The XOR approach is slightly slower than the original BD protocol, but the tiny performance penalty is insignificant in practice.[4] The Simple KLL protocol does not provide authentication or public verifiability, because these properties are implemented in a higher protocol layer; these properties are revisited in Section 9.4.

When a protocol description in this chapter indicates that a participant "broadcasts" a message, this refers to the participant transmitting the message to all other participants using a reliable transmission mechanism. When implemented in practice over the Internet, this normally means that the participant sends the message to a central server that in turn retransmits the message to the other participants, all using a reliable TCP-based network protocol. If a participant fails to successfully broadcast a message, then the protocol will time out and abort; Safehouse is designed to accommodate this potential outcome. It is important to note that this definition of "broadcast" is not as strong as the definition used by *broadcast protocols* [CKPS01], which additionally require that each participant is assured that the same message was delivered to all other participants. The authenticated version of TKLL introduced in Section 9.4 will detect misbehavior such as equivocation (i.e., if a malicious participant sends different messages to different recipients) and abort the protocol in response.

[4] ^ The BD protocol requires one scalar inversion and one scalar multiplication to "mix" the shared secrets, whereas KLL requires two scalar multiplications. For groups on elliptic curves like Curve25519, inversions are marginally faster than scalar multiplications, but not by enough to matter in practice for this use case.

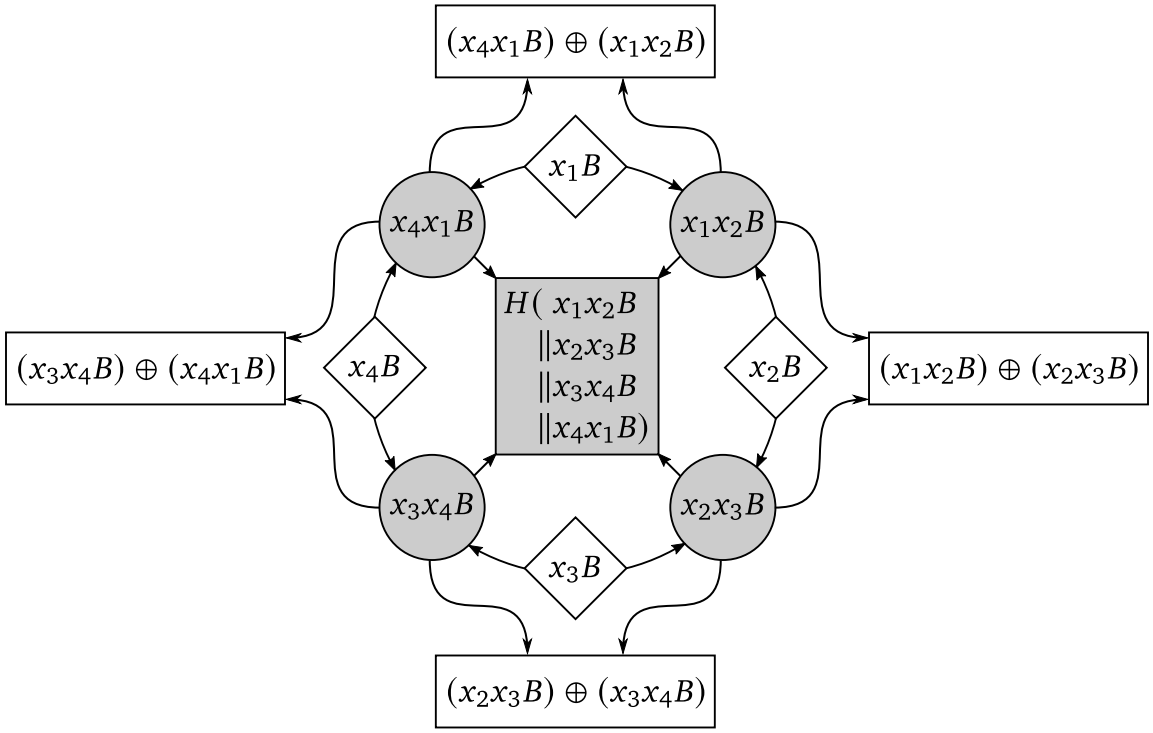


Figure 9.2 A SIMPLE KLL KEY EXCHANGE WITH FOUR PARTICIPANTS. Shaded values are internally computed and are not sent over the network. Given knowledge of all unshaded values and any one of the DH private keys in $\{x_1, x_2, x_3, x_4\}$, one can compute the group key (depicted in the central shaded rectangle). (Refs: 296 [a] [b])

Simple KLL establishes a shared secret in only two rounds of communication:

1. The participants logically arrange themselves into a circle. All participants broadcast a DH public key. Upon receipt of all other keys, each participant computes the DH shared secret with their two neighbors in the circle; these are the "left" and "right" secrets for the participant.

2. All participants broadcast the XOR of their "left" and "right" secrets. Upon receipt of this information, each participant can recover *all* of the DH shared secrets by using one of their own shared secrets to recursively "undo" a neighbor's XOR. The group key is a KDF of all of the DH shared secrets.

Figure 9.2 depicts the keys involved in this procedure for a Simple KLL performed between four participants. From the perspective of a single participant (U_1), the protocol example in Figure 9.2 proceeds as follows:

1. The four participants are unambiguously labeled as U_1 through U_4 as input to the protocol. This example is written in the perspective of participant U_1, but the other cases are symmetric.

2. U_1 generates a random scalar x_1 in the field for the elliptic curve group, then computes the DH public key $X_1 = x_1 B$.

3. U_1 sends X_1 to U_2 and U_4. U_1 receives X_2 from U_2 and X_4 from U_4.[5] The points are checked to ensure that they are in the elliptic curve group.

4. U_1 computes the "left" secret $t_{4,1} = x_1(x_4 B)$ and the "right" secret $t_{1,2} = x_1(x_2 B)$. It then computes $Y_1 = t_{4,1} \oplus t_{1,2}$. In practice, this involves compressing the two elliptic curve points into a binary encoding before computing the XOR operation.

5. U_1 broadcasts Y_1 and receives Y_2, Y_3, and Y_4 in return.

6. U_1 computes all unknown DH shared secrets as follows:

   a) $t_{2,3} = Y_2 \oplus t_{1,2}$.

   b) $t_{3,4} = Y_3 \oplus t_{2,3}$.

   c) As a sanity check, U_1 ensures that $t_{4,1} = Y_4 \oplus t_{3,4}$.

7. U_1 computes the group key $H(t_{1,2} \| t_{2,3} \| t_{3,4} \| t_{4,1})$, where H denotes the KDF function.

[5] In Simple KLL, it is only necessary for participants to exchange DH public keys with their neighbors. When the protocol is used in the authenticated version of TKLL, the DH public keys must be broadcast to all other participants in the Simple KLL exchange, because all of the DH public keys will be digitally signed.

Algorithm 9.1 SIMPLE KLL STEP 1. A new DH key pair is generated. (Refs: 297 and 313)

Subroutine | Simple-KLL$_1(i, n) \rightarrow (l, r, x_i, X_i)$

if $(i > 1)$ { Let $l = i - 1$ } **else** { Let $l = n$ }
if $(i < n)$ { Let $r = i + 1$ } **else** { Let $r = 1$ }
Choose $x_i \xleftarrow{\$} [2, q_E)$.
$X_i \leftarrow x_i \cdot B$.
return (l, r, x_i, X_i).

The computations performed by the other participants are symmetric.

The precise steps for Simple KLL can be written as three generalized subroutines. The first step is to compute the caller's position in the circle and to generate a DH key pair in the elliptic curve group. This procedure is defined in Algorithm 9.1. Simple-KLL$_1$ takes as input the number of participants n and the caller's participant number $i \in [1, n]$. It outputs the indices of the "left" and "right" neighbors (l and r, respectively), the DH private key x_i, and the DH public key X_i.

The next step is to compute the "left" and "right" secrets and the XOR of their compressed forms. This procedure is defined in Algorithm 9.2. In practice, the inputs l, i, r, and x_i are the same as in Simple-KLL$_1$, and the neighbors' DH public keys X_l and X_r are received over the network. The function outputs the secrets shared with the "left" and "right" neighbors ($t_{l,i}$ and $t_{i,r}$, respectively) and their XOR value Y_i. Internally, Simple-KLL$_2$ makes use of a function called Compress$_E(P) \rightarrow D$. This function takes a point P on the elliptic curve E as input and outputs D, a binary representation of this point. The inverse function is Decompress$_E(D) \rightarrow P$.

The final step is to compute all of the other DH secrets and to compute the group key. This procedure is defined in Algorithm 9.3. The input $\vec{Y}$ is a vector of all n XOR values; $n - 1$ of these must be received from the other participants. Y_i refers to the XOR value produced by participant $i \in [1, n]$. In practice, the other inputs to the function are the same as in the previous steps. The output of the function, z, is a shared secret called the *group key preimage* that is learned by all participants. To produce the *group key*, the group key preimage is given as input to H, the KDF. Note that the definition of z does not include the secret $t_{n,1}$. Including this extra term is not necessary for security. Omitting the term causes the scheme to reduce to a standard DH key exchange in the event that $n = 2$. This property simplifies the construction of TKLL.

The prototype implementation instantiates the elliptic curve E with Curve25519 [Ber06]. The implementation performs ECDH on Curve25519 using a standard X25519 library for Go. X25519 defines several security and performance optimizations, including storing points as only x coordinates. This lossy compression method drops the y coordinate, but this does not affect the correctness of ECDH. This is also not a problem for Simple KLL, since it is not necessary for

Algorithm 9.2 SIMPLE KLL STEP 2. The "left" and "right" DH shared secrets are computed and the XOR of their compressed forms is derived. (Refs: 297, 315, and 318)

Subroutine | $\text{Simple-KLL}_2(l, i, r, x_i, X_l, X_r) \rightarrow (t_{l,i}, t_{i,r}, Y_i)$

$t_{l,i} \leftarrow x_i \cdot X_l$.
$t_{i,r} \leftarrow x_i \cdot X_r$.
$Y_i \leftarrow \text{Compress}_E(t_{l,i}) \oplus \text{Compress}_E(t_{i,r})$.
return $(t_{l,i}, t_{i,r}, Y_i)$.

Algorithm 9.3 SIMPLE KLL STEP 3. All of the DH shared secrets are computed and the group key preimage z is derived. The group key is $H(z)$. (Refs: 297 and 318)

Subroutine | $\text{Simple-KLL}_3(l, i, n, \vec{Y}, t_{l,i}, t_{i,r}) \rightarrow (z)$

for $(j \in [0, n-1)$ in ascending order) {
  if $(i + j \leq n)$ { Let $a = i + j$ } **else** { Let $a = i + j - n$ }
  if $(a < n)$ { Let $b = a + 1$ } **else** { Let $b = 1$ }
  if $(b < n)$ { Let $c = b + 1$ } **else** { Let $c = 1$ }
  if $(b \neq l)$ {
    $t_{b,c} \leftarrow \text{Decompress}_E(\text{Compress}_E(t_{a,b}) \oplus Y_b)$.
  } **else** {
    $\overline{t_{l,i}} \leftarrow \text{Decompress}_E(\text{Compress}_E(t_{a,l}) \oplus Y_l)$.
  }
}
if $(\overline{t_{l,i}} \neq t_{l,i})$ Abort with $z = \perp$.
$z \leftarrow \text{Compress}_E(t_{1,2}) \| \text{Compress}_E(t_{2,3}) \| \ldots \| \text{Compress}_E(t_{n-1,n})$.
return z.

Decompress_E to be a true inverse of Compress_E: it is only necessary for all participants to recover the same *compressed* DH secrets (and therefore the same value of z). Formally, it is only necessary that:

$$\forall_{P \in E}\ \text{Compress}_E(\text{Decompress}_E(\text{Compress}_E(P))) = \text{Compress}_E(P)$$

It is easy to see that Simple KLL provides *key secrecy* against passive network adversaries when the CDH assumption holds in the ROM. Specifically, the adversary cannot distinguish $H(z)$ from a random value of the appropriate length when H is modeled by a random oracle: doing so would require knowledge of z, which is a concatenation of all DH shared secrets. The same game-based approach used by Kim et al. [KLL04] to prove that KLL is secure can be used to reduce an adversary that can recover z to a CDH adversary.

9.3 Unauthenticated Tree Kim-Lee-Lee

This section describes an unauthenticated version of the TKLL protocol. The main purpose of TKLL is to interactively establish a given key tree using as few rounds of communication as possible. The structure of the key tree is given as an input to TKLL. When TKLL is used as part of Safehouse, the protocol for "mass join" operations determines the shape of the key tree that TKLL must instantiate. The key tree is structured so that the resulting keys can be used to efficiently add the participants into a group conversation as part of a cryptographic operation that is sent to existing group members non-interactively.

Each TKLL participant can generate the key pair for their own personal key in the given key tree independently. Internal k-nodes in the tree with more than one child are more difficult to establish: the corresponding private key must be learned by every participant in the node's user set (i.e., every participant with a u-node in the subtree). TKLL accomplishes this by performing Simple KLL (as defined in Section 9.2) between one representative participant for each child k-node; these representatives all learn the private key for the k-node.

In order for *all* participants in a user set to learn the private key used by their representative, each representative sends the DH private key that they used in the Simple KLL protocol to all of the other participants in their subtree—this allows the other participants to follow along with the Simple KLL execution. To preserve security, this DH private key is encrypted to one of the DH shared secrets in the representative's subtree using a symmetric cryptosystem, where Enc denotes the encryption function and Dec denotes the decryption function; correctness ensures that $\text{Dec}(\kappa, \text{Enc}(\kappa, m)) = m$ for a key κ and message m. The shared secret that is used as the key for the ciphertext is known to all of the intended recipients, and it is already known to the

representative after the first round of communication; this allows the ciphertext to be sent as part of the second communication round. A KDF denoted KDF_1 transforms the shared secret into a key suitable for use by Enc.

There are two important edge cases when generating key pairs for internal k-nodes: k-nodes with exactly one or two children. When a k-node has only one child, the next step depends on whether the child is the participant's personal key or not. If the k-node's only child is not the participant's personal key, then the group key preimage for the k-node is derived from the child's group key preimage using a KDF denoted KDF_2. If the k-node's only child is the participant's personal key, then the group key preimage for the k-node is generated randomly. When a k-node has two children, the Simple KLL protocol reduces to a two-party DH exchange, as discussed in Section 9.2. In this case, it is not necessary for the representatives to actually send the XOR value computed in Simple-KLL_2 over the network, since the value will always be 0. When the k-node has more than two children, the protocol proceeds as in the general case.

Once the key exchanges are complete, each k-node will have an associated group key preimage generated by Simple KLL, derived from another group key preimage with KDF_2, or generated randomly, based on the position of the k-node in the tree. The group key preimages produced are given as input to H (the KDF defined in Section 9.2) to derive group keys for the k-nodes. These group keys are then provided as seeds for the key generation function of the target cryptosystem (e.g., the key space used by a BRAKEM scheme defined in Chapter 8). This key generation function, denoted by $\text{KeyGen}(s)$, takes a seed s as input and deterministically produces a key pair (pk, sk) from the seed as output. Passing all of the group key preimages through H before providing them to KeyGen ensures that the resulting private keys are independent. Using this procedure, key pairs for the entire tree can be generated after only two communication rounds, regardless of the tree's size or complexity. In a third and final communication round, the representative for each k-node sends its public key to participants that are not in its user set.

A formal definition of TKLL is postponed until Section 9.4, since the inclusion of authentication mechanisms changes several important protocol details. In this section, the unauthenticated form of TKLL is best understood through a specific example. Figure 9.3 depicts a sample key tree that may be used as an input to TKLL. Given an ordered list of participants $U_1, \ldots, U_6$ (with corresponding u-nodes labeled $u_1, \ldots, u_6$) and a target key generation function $\text{KeyGen}(s)$, the six participants are expected to instantiate the key tree with a key pair for each k-node. To summarize the prior description of the protocol, the participants will accomplish this feat in three rounds of communication:

1. For each k-node that is not a personal key (k'_1, k''_1, k_{234}, k_{56}, k'_{56}, and k_{123456} in the example) and has at least two children (k_{234}, k_{56}, and k_{123456} in the example), the leftmost participant in each child node's user set is unambiguously chosen as the representative for that child

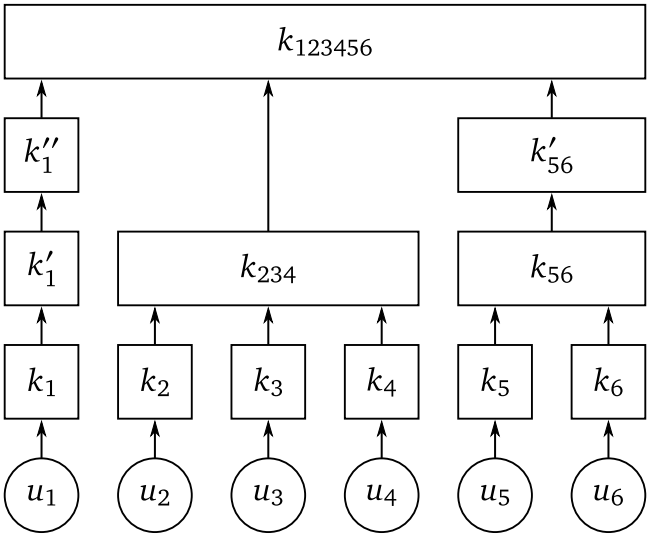


Figure 9.3 A SAMPLE KEY TREE USED AS INPUT TO UNAUTHENTICATED TKLL. The rectangles are k-nodes and the circles are u-nodes. (Refs: 300, 302, 303, and 304)

node. For example, in the GKE that will establish the key for k_{123456}, the representatives for the subtrees rooted at k''_1, k_{234}, and k'_{56} are U_1, U_2, and U_5, respectively. For each k-node that is not a personal key and has at least two children, the representatives perform Simple-KLL$_1$ and broadcast their newly generated DH public keys to all participants in the user set for the k-node. Although the representatives are responsible for computing and sending the messages of the GKE, all participants in the subtree rooted at the node for the GKE (not only the representatives) must receive the messages, because they must follow along with the progress of the GKE. For example, U_2 will broadcast its new DH public key in the exchange for k_{234} to U_3 and U_4. U_2 will additionally broadcast its (distinct) new DH public key in the exchange for k_{123456} to the five other participants. Similar broadcasts are made by the other participants.

2. In each GKE, the representatives, having received the DH public keys from the other representatives, each perform Simple-KLL$_2$ and broadcast the newly computed XOR values to the other participants in the user set for the k-node. Moreover, each representative for a child node that has participants other than the representative in its user set must also encrypt its DH private key from the first round to those other participants. The representative traverses down the subtree to find the shallowest k-node (i.e., the closest k-node to the root) with two or more children—this will be the k-node in the represented subtree that is the closest to the root and has also completed the first round of a Simple KLL GKE. The DH private key

is encrypted to the "first" DH shared secret in the circle for this Simple KLL protocol, which will always be available to the representative at the time that the ciphertext is created. For example, U_2 will encrypt the DH private key it used in the GKE for k_{123456} to the DH shared secret between U_2 and U_3 in the GKE for k_{234}; U_3 can decrypt this ciphertext immediately, whereas U_4 will be able to decrypt it once the round is complete. As another example, U_5 will encrypt the DH private key it used in the GKE for k_{123456} to the DH shared secret between U_5 and U_6 in the GKE for k_{56}. These are the only two ciphertexts that are sent for the key tree in this example.

3. Each participant traverses the tree in order from their u-node to the root of the tree. For each k-node with multiple children on the path, the participants execute Simple-KLL$_3$ to compute the group key preimage. When a k-node with only one child is encountered during the traversal and the child is not a personal key, the group key preimage for the node is computed by applying a KDF denoted KDF$_2$ to the child node's group key preimage. When a k-node with only one child is encountered and the child is a personal key, the participant generates the group key preimage randomly. In any case, if the participant is represented by another participant in the GKE for the parent node, then the ciphertext sent in the previous step is decrypted using the newly recovered "first" DH shared secret. In this example, U_1 generates the key for k'_1 randomly. The key for k''_1 is derived from the key for k'_1 with KDF$_2$, and the key for k'_{56} is similarly derived from the key for k_{56}. After these operations, group key preimages have been established for all of the k-nodes. For each group key preimage z, participants in the user set compute the associated key pair using KeyGen$(H(z)) \rightarrow (pk, sk)$. For each k-node, the "leftmost" participant in the user set broadcasts the new public key to all participants in the protocol (e.g., U_5 broadcasts the final public key for k'_{56} to the other five participants). The key tree is now established.

Figure 9.4 depicts all of the keys that are involved in unauthenticated TKLL for the sample key tree T in Figure 9.3. The notation for the trees is the same as in the Simple KLL functions defined in Section 9.2, but with added superscripts that indicate which node in the key tree they apply to. A superscript "(v)" indicates that the value applies to the GKE for parent$_T(v)$. Using the child node in the superscript simplifies the pseudo-code for TKLL. Since this superscript is different for each participant based on which child subtree their u-node belongs to, Figure 9.4 depicts all possible superscripts for each value separated by vertical bars. For each Simple KLL execution, the indices of the participants are expressed relative to the user set for the node instead of in absolute terms. For example, the values associated with U_2 in the GKE for k_{234} are given the index 1, since U_2 is the "first" participant in userset$_T(k_{234})$. To summarize the notation:

- $x_i^{(v)}$ denotes the DH private key generated by Simple-KLL$_1$ for participant i in the GKE for k-node parent$_T(v)$, and $X_i^{(v)}$ is the associated DH public key.

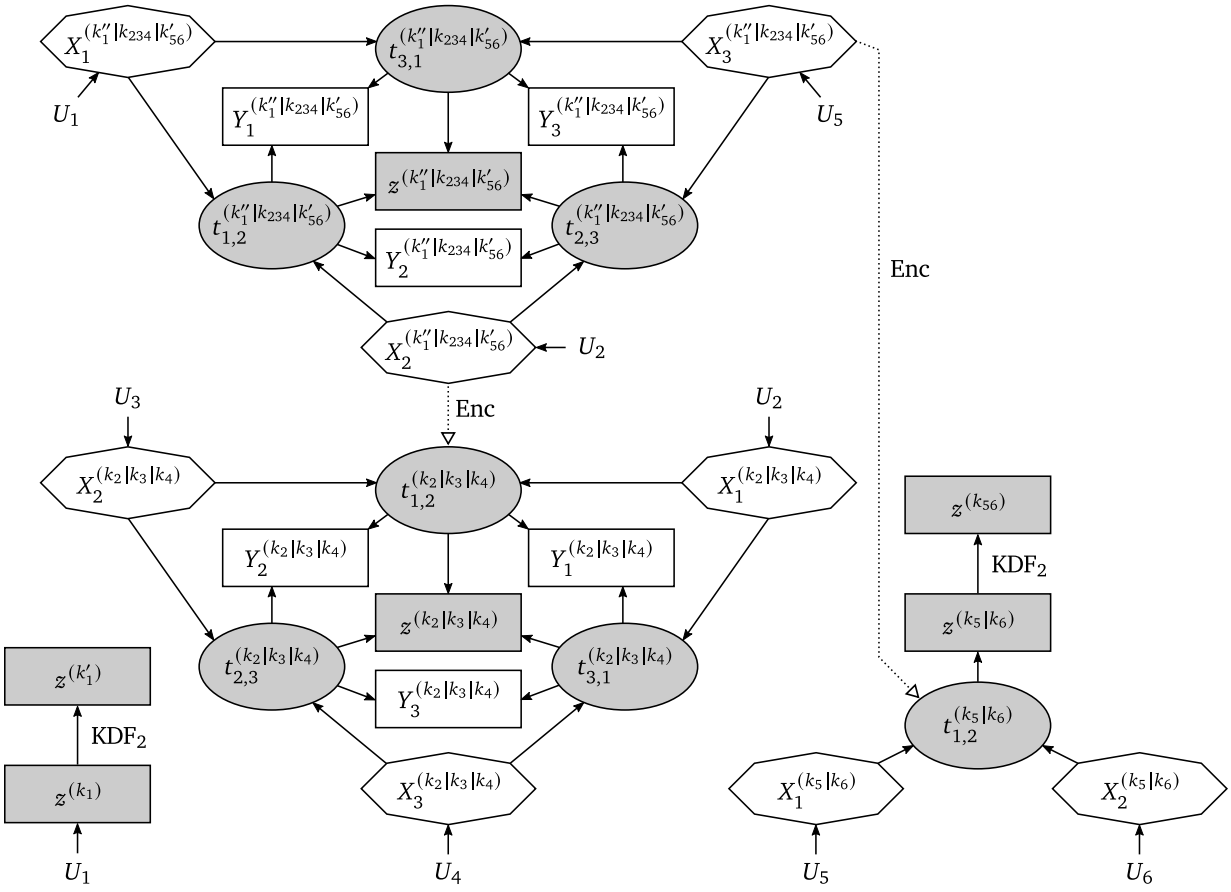


Figure 9.4 AN EXAMPLE OF KEYS INVOLVED IN UNAUTHENTICATED TKLL EXECUTION. These values are generated by the protocol to establish the key tree in Figure 9.3. Shaded values are not sent over the network. Arrows indicate data dependencies. An arrow from $U_1, \ldots, U_6$ indicates that the participant generates the specified key. Superscripts of the form $(k_1|k_2)$ indicate that the value might have superscript k_1 or k_2 depending on the perspective: each client uses different superscripts for keys in its local state based on the position of its u-node in the tree. (Refs: 302[a][b] and 304)

- $t_{i,j}^{(v)}$ denotes the DH shared secret between $X_i^{(v)}$ and $X_j^{(v)}$.
- $Y_i^{(v)}$ denotes the XOR of DH shared secrets for participant i's left and right neighbor indices l and r: $\text{Compress}_E\left(t_{l,i}^{(v)}\right) \oplus \text{Compress}_E\left(t_{i,r}^{(v)}\right)$.
- $z^{(v)}$ is the group key preimage for k-node $\text{parent}_T(v)$. Based on the key tree structure, it is either generated randomly, derived using KDF_2, or output by Simple-KLL_3.
- $sk^{(v)}$ is the private key for k-node $\text{parent}_T(v)$. It is computed using $\text{KeyGen}\left(H\left(z^{(v)}\right)\right)$. The associated public key is $pk^{(v)}$.

It can be illuminating to work through the example from the perspective of a single participant. From the perspective of U_6, the example in Figures 9.3 and 9.4 proceeds as follows:

1. U_6 generates $\left(x_2^{(k_6)}, X_2^{(k_6)}\right)$ using Simple-KLL_1. It sends $X_2^{(k_6)}$ to U_5.
2. U_6 receives $X_1^{(k_6)}$ from U_5. It also receives $X_1^{(k'_{56})}$ from U_1, $X_2^{(k'_{56})}$ from U_2, and $X_3^{(k'_{56})}$ from U_5.
3. U_6 computes $t_{1,2}^{(k_6)}$ using Simple-KLL_2. U_6 has no XOR values to broadcast because it is not the representative for any k-node with more than two children.
4. U_6 receives $Y_1^{(k'_{56})}$ from U_1, $Y_2^{(k'_{56})}$ from U_2, and $Y_3^{(k'_{56})}$ from U_5. It also receives $\text{Enc}\left(\kappa, x_3^{(k'_{56})}\right)$ from U_5, which is U_6's representative in the GKE for k_{123456}, where $\kappa = \text{KDF}_1\left(t_{1,2}^{(k_6)}\right)$. $\text{Enc}(\kappa, m)$ denotes a symmetric encryption of message m for key κ.
5. U_6 computes $\text{KDF}_1\left(t_{1,2}^{(k_6)}\right)$ and uses it to decrypt $x_3^{(k'_{56})}$. It then uses Simple-KLL_2 to compute $t_{2,3}^{(k'_{56})}$ and $t_{3,1}^{(k'_{56})}$. Using the XOR values and the DH shared secrets, it uses Simple-KLL_3 to compute $z^{(k_6)}$ and $z^{(k'_{56})}$.
6. U_6 computes $\text{KDF}_2\left(z^{(k_6)}\right) \rightarrow z^{(k_{56})}$.
7. U_6 computes the final key pairs:
   - $\text{KeyGen}\left(H\left(z^{(u_6)}\right)\right) \rightarrow \left(pk^{(u_6)}, sk^{(u_6)}\right)$ for $z^{(u_6)}$ chosen randomly.
   - $\text{KeyGen}\left(H\left(z^{(k_6)}\right)\right) \rightarrow \left(pk^{(k_6)}, sk^{(k_6)}\right)$.

- $\text{KeyGen}\left(H\left(z^{(k_{56})}\right)\right) \rightarrow \left(pk^{(k_{56})}, sk^{(k_{56})}\right)$.
- $\text{KeyGen}\left(H\left(z^{(k'_{56})}\right)\right) \rightarrow \left(pk^{(k'_{56})}, sk^{(k'_{56})}\right)$.

U_6 broadcasts $pk^{(u_6)}$ to the other five participants.

8. U_6 receives all other public keys from the other participants:
   - U_1 sends the public keys for k_1, k'_1, and k''_1.
   - U_2 sends the public keys for k_2 and k_{234}.
   - U_3 sends the public key for k_3.
   - U_4 sends the public key for k_4.
   - U_5 sends the public key for k_5.

9.4 Authenticated Tree Kim-Lee-Lee

This section augments the unauthenticated TKLL protocol described in Section 9.3 with authentication mechanisms, yielding the final protocol definition.

9.4.1 Design Overview

Authentication mechanisms for GKEs in the BD family (such as Simple KLL) were previously described in Section 3.4. The simplest approach would be to apply the Katz-Yung compiler [KY03] to TKLL, digitally signing all messages with a long-term key; this is essentially the approach used by Dutta and Barua [DB08]. This method is inefficient and is not general enough for the intended use in Safehouse. The original BD protocol [BD94] essentially included a NIZKPK in the first message flow to prove knowledge of the DL of both the ephemeral DH private key *and* a long-term private key. KLL [KLL04] and the protocol from Yang and Tan [YT10] include "keying bytes" to achieve contributiveness, and use long-term keys to digitally sign both the keying bytes and the ephemeral DH public keys—this provides authentication and defends against active network adversaries.

TKLL's authentication mechanisms differ from all of the aforementioned protocols. The mechanisms must achieve three goals: establish and authenticate long-term identities, defend

against active network adversaries, and prevent insider attacks such as equivocation (where a participant causes other participants to derive different group keys).

The authentication mechanism to establish and authenticate long-term identities varies based on the Safehouse configuration. Specifically, different mechanisms must be used based on whether deniability or non-repudiation is desired. Since this mechanism is defined by the higher-level protocol (i.e., Safehouse) but the data interacts with the rest of the TKLL protocol, TKLL is parameterized by two functions that handle identification and authentication:

- Compute-ID$(P^*, Q^*) \rightarrow I$ generates a long-term identity I once TKLL has established the key tree T. P^* is a mapping from k-nodes in T to public keys, and Q^* is a mapping from k-nodes in T to private keys. P^* and Q^* are only required to have mappings for k-nodes on the path to the caller's u-node; this is sufficient for the intended application, and it allows long-term identities to be established in an early communication round.
- Prove-ID$(I^*, P^*, Q^*) \rightarrow \pi$ produces a NIZKPK that authenticates the calling participant. P^* and Q^* are mappings for public keys and private keys in the tree, as in the definition for Compute-ID. I^* is a mapping from participants to their identities, as produced by Compute-ID. The output π is the proof that authenticates the caller's identity.

Defending against active network adversaries is accomplished using a digital signature scheme. The approach is similar to the outcome of applying the Katz-Yung compiler to TKLL. Unlike the unauthenticated version of TKLL described in Section 9.3, the authenticated version does not internally generate key pairs for the participants' personal keys. The public keys for all participants' personal keys and the private key for the caller's personal key are accepted as an input to the algorithm; this allows the personal key pairs to be used for other authentication tasks prior to the invocation of TKLL. The personal key pairs are used with a digital signature scheme to authenticate TKLL transmissions. As part of the second round of TKLL, each participant signs all of the other data that it sent in the first two rounds using its personal private key. In particular, the DH public keys and XOR values are signed. Participants verify these signatures upon receipt. Assuming that the personal public keys were distributed to all participants correctly, these signatures prevent attacks by active network adversaries, since any modification of the transmitted data would require forging a signature. TKLL is parameterized by a signing function $\text{Sig}(pk, sk, m) \rightarrow \sigma$ that produces a digital signature σ for message m using the personal key pair (pk, sk), and a corresponding verification function $\text{SVerif}(pk, m, \sigma) \rightarrow \{0, 1\}$ that outputs 1 if and only if the signature σ on message m is valid for the public verification key pk.

In general, this means that the authenticated form of TKLL involves keys from three different public-key cryptosystems:

1. The DH key pairs generated as part of the Simple KLL subprotocol come from a group in which DH key exchanges are secure and efficient, such as Curve25519 [Ber06].

2. The key pairs associated with personal k-nodes come from a group defined by the higher-level protocol. These keys are used in the digital signature scheme defined by the Sig and SVerif functions provided by the higher-level protocol. As discussed later in this section, these keys are also used in a multi-signature scheme.

3. The key pairs associated with internal k-nodes belong to the target key space. These keys are generated by the KeyGen function provided by the higher-level protocol, which takes as input a seed produced by the H KDF.

In general, these three key spaces can be distinct. When TKLL is used by Safehouse in practice, the second and third key spaces (associated with personal and internal k-nodes) are the same key space used by the BRAKEM construction; key pairs from this space are used both as BRAKEM encapsulation recipients and to produce digital signatures.

To defend against insider attacks such as equivocation, it is necessary to ensure that all participants derive the same final state. The easiest way to accomplish this is to have each participant digitally sign the final state (i.e., the key tree structure, the final public keys for the k-nodes, and the list of participants). While it would be possible to do this using a standard digital signature scheme, the interactive nature of TKLL enables the use of a *multi-signature*: an interactively generated signature of constant size that attests to a message being signed by all of the participants. TKLL uses the MS-BN multi-signature scheme proposed by Bellare and Neven [BN06]. MS-BN was chosen because it is a simple and highly efficient three-round multi-signature scheme that is suitable for use with the key spaces defined by the BRAKEM constructions in Chapter 8, it fits easily within TKLL's communication rounds, and it does not require additional security assumptions or additional key generation steps.[6] The MS-BN implementation used in TKLL must operate in the same key space as the one for the Sig and SVerif digital signature scheme (i.e., the key space for personal keys), which is defined by the higher-level protocol.

[6] ^ There are many newer multi-signature schemes that provide superior features compared to MS-BN, such as MuSig [MPSW19], MuSig2 [NRS20], and FROST [KG20], among others. However, all of these schemes provide improved functionality by sacrificing performance or security in some regard, and none of the improved functionality is necessary in this setting: MuSig and MuSig2 both require the "one-more discrete logarithm" security assumption in the ROM, rather than the DL assumption already required by BRAKEM and TKLL; MuSig2 additionally requires the transmission of $O(N^2)$ group elements (compared to $O(N)$ group elements for MS-BN), where N is the number of signers, in order to complete one round sooner; and FROST is a more general threshold signature scheme that requires an interactive key setup. Given the simplicity of MS-BN and the lack of improvement in its exact setting over the past 15 years, it seems likely that MS-BN will remain the best option for use in TKLL for the foreseeable future.

Consequently, TKLL is additionally parameterized by the following functions that implement MS-BN in the target key space:

- $\text{MSBN.Sign}_1() \to (r_1, R_1, t_1)$ implements round 1 of MS-BN. This function generates a random DH key pair with private key r_1 and public key R_1, as well as a hash of R_1 called t_1. The value t_1 is expected to be sent to the other participants.
- Round 2 of MS-BN is implemented within TKLL and does not need to be provided as a function by the higher-level protocol. In round 2, the participant sends R_1 to each other participant i after receiving t_i.
- $\text{MSBN.Sign}_3(\vec{L}, \vec{t}, \vec{R}, r_1, x_1, m) \to (s_1)$ implements round 3 of MS-BN. $\vec{L}$ is a sequence of public keys for the participants, including the caller, in arbitrary order.[7] $\vec{t}$ is a sequence of t values received from the other participants, with t_i being received from participant L_i. Likewise, $\vec{R}$ is a sequence of R values with R_i being received from participant L_i. r_1 is the secret generated by MSBN.Sign_1, and x_1 is the private key corresponding to L_1. m is the message to sign. The output of the function is a response s_1 that should be sent to the other participants, or $\perp$ if the protocol should be aborted due to invalid input.
- $\text{MSBN.Sign}_4(\vec{L}, \vec{R}, \vec{s}) \to (\sigma)$ implements the final step of MS-BN in which the multi-signature is computed. $\vec{L}$ and $\vec{R}$ are defined as in MSBN.Sign_3. The sequence $\vec{s}$ contains s values such that s_i was received from participant L_i. The output of the function is a signature σ that attests to a message being signed by all participants in $\vec{L}$.
- $\text{MSBN.Vf}(\vec{L}, m, \sigma) \to (b)$ verifies an MS-BN multi-signature σ. It returns $b = 1$ if σ is a valid signature produced by all private keys corresponding to the public keys in $\vec{L}$ on message m, or $b = 0$ otherwise.

It is relatively straightforward to implement these functions for a given key space using Bellare and Neven's definitions [BN06, §5]. For example, Algorithm 9.4 shows how to implement the functions for the $\text{BRAKEM}^{\text{DDL}}$ key space using the group notation defined in Section 8.2.1.

To summarize, authentication in TKLL is achieved with three mechanisms: the externally defined identification and authentication functions, digital signatures on the Simple KLL transmissions, and a multi-signature on the final state of the key tree. The unauthenticated version of TKLL in Section 9.3 used three rounds of communication. The Compute-ID function cannot

[7] ^ MS-BN does not require that participants prove knowledge of their private keys. However, this occurs implicitly in TKLL in practice, since all participants produce and broadcast digital signatures using their personal key pairs prior to the completion of the MS-BN protocol.

Algorithm 9.4 MS-BN MULTI-SIGNATURE FUNCTIONS FOR THE BRAKEM$^{\text{DDL}}$ KEY SPACE. H_0 and H_1 are distinct cryptographic hash functions modeled by random oracles.
(Refs: 308, 310, 313, 320, and 321 [a] [b])

Function | MSBN.Sign$_1$() $\rightarrow (r_1, R_1, t_1)$

Choose $r_1 \xleftarrow{\$} [2, p_3)$.
Compute $R_1 \leftarrow {g_2}^{r_1}$.
Compute $t_1 \leftarrow H_0(R_1)$.
return (r_1, R_1, t_1).

Function | MSBN.Sign$_3(\vec{L}, \vec{t}, \vec{R}, r_1, x_1, m) \rightarrow (s_1)$

for $(2 \leq i \leq |\vec{L}|)$ {
  if $(t_i \neq H_0(R_i))$ { **return** $\perp$ }
}
Compute $R \leftarrow \prod_{i=1}^{|\vec{L}|} R_i \pmod{p_2}$.
Compute λ, a unique encoding of $\vec{L}$.
Compute $c_1 \leftarrow H_1(L_1 \| R \| \lambda \| m)$.
Compute $s_1 \leftarrow x_1 c_1 + r_1 \pmod{p_3}$.
return s_1.

Function | MSBN.Sign$_4(\vec{L}, \vec{R}, \vec{s}) \rightarrow (\sigma)$

Compute $R \leftarrow \prod_{i=1}^{|\vec{L}|} R_i \pmod{p_2}$
  $\hookrightarrow$ (MSBN.Sign$_3$ can cache R for efficiency.)
Compute $s \leftarrow \sum_{i=1}^{|\vec{L}|} s_i \pmod{p_3}$.
return $\sigma = (R, s)$.

Function | MSBN.Vf$(\vec{L}, m, \sigma) \rightarrow (b)$

Interpret $\sigma = (R, s)$, returning 0 if decoding fails.
Compute λ, a unique encoding of $\vec{L}$.
for $(1 \leq i \leq |\vec{L}|)$ {
  Compute $c_i \leftarrow H_1(L_i \| R \| \lambda \| m)$.
}
Compute $A = {g_2}^s$.
Compute $B = R \cdot \left(\prod_{i=1}^{|\vec{L}|} {L_i}^{c_i} \right)$.
if $(A \neq B)$ { **return** 0 }
return 1.

produce a long-term identity until key pairs have been established for all k-nodes on the path to the calling participant's u-node. This occurs after the second round, so the long-term identities are generated and broadcast during the third round. Since Prove-ID requires the identities for all participants, the authenticating NIZKPKs are generated and broadcast during a fourth round. The digital signatures for the DH public keys and XOR values can be sent in the second round, which is the earliest that these values are all known. These signatures are sufficient to protect key secrecy against active network adversaries. The multi-signatures cannot be produced until the key tree has been fully established. This occurs at the end of the third round, after the public keys for all k-nodes have been distributed. Consequently, MSBN.Sign_4 is called after the third round, and the results are broadcast in the fourth round. The other parts of the multi-signature scheme must take place earlier. The hashes produced by MSBN.Sign_1 are distributed in the first round, the commitments are distributed in the second round, and the responses produced by MSBN.Sign_3 are distributed in the third round. This completes the changes required to implement the authentication mechanisms.

9.4.2 Formal Definition

The authenticated form of TKLL takes the following input parameters:

- $\overrightarrow{ID}$: a sequence of numeric identifiers for the participants with the calling participant's identifier in the first position.
- $\vec{L}$: a sequence of personal public keys for the participants, in the same order as $\overrightarrow{ID}$.
- T: the key tree to instantiate. The personal keys in T will be populated with the keys in $\vec{L}$.
- sk: the private key corresponding to L_1.
- An identification function Compute-ID and an authentication function Prove-ID.
- An implementation of a digital signature scheme given as two functions: Sig and SVerif.
- An implementation of MS-BN given as the following functions: MSBN.Sign_1, MSBN.Sign_3, MSBN.Sign_4, and MSBN.Vf (see Algorithm 9.4 for an example).

The function parameters are given implicitly; they are hard-coded in a real implementation with a fixed key space. Participant i is denoted U_i, where the index i refers to the same order as $\overrightarrow{ID}$. The output from TKLL contains the following values:

- P^*: a mapping from k-nodes to public keys.
- Q^*: a mapping from k-nodes to private keys, with one entry for every k-node on the path to the u-node for U_1.
- I^*: a mapping from participants to their identities, as produced by Compute-ID.
- Π^*: a mapping from participants to NIZKPKs authenticating their identities.
- σ: a multi-signature proving that all participants accept the instantiation of the key tree given by the other outputs.

Once TKLL has been performed, a "ciphertext" message containing $(T, P^*, I^*, \Pi^*, \sigma)$ can be prepared. This message can be sent to a set of recipients non-interactively. Once the recipients verify that σ and the proofs in Π^* are correct, they can "import" the key tree T with public keys P^* and identities I^* into their state. This is the basis for the "mass join" operation in Safehouse.

TKLL requires four rounds of communication. Each round has a "prep" function that prepares the required state, a "send" function that sends the required broadcasts for the round, and a "receive" function that updates the state with the broadcasts received from other participants. The rounds perform the following operations:

1. The first round of Simple KLL is performed for relevant k-nodes. The first round of MS-BN is performed.
2. The second round of Simple KLL is performed. The second round of MS-BN is performed. Representatives encrypt DH private keys to their subtrees.
3. The final key pairs are derived for each k-node. Representatives broadcast public keys for k-nodes to participants in other subtrees. The third round of MS-BN is performed. Identities are generated and broadcast.
4. The fourth round of MS-BN is performed. Identity NIZKPKs are generated and broadcast.

The TKLL protocol is fully specified in Algorithm 9.5. The functions for each round are defined by individual algorithms: round one in Algorithms 9.6, 9.7, and 9.8; round two in Algorithms 9.9, 9.10, and 9.11; round three in Algorithms 9.12, 9.13, and 9.14; and round four in Algorithms 9.15, 9.16, and 9.17. The generalized version of TKLL given by these algorithms also works correctly for the edge case in which there is only one participant; T must be a path graph in this case. When there is only one participant, that participant unilaterally generates key

pairs for all k-nodes without any transmissions and signs the result using the digital signature scheme instead of MS-BN. For clarity, the TKLL pseudocode omits some validation of incoming data. Real implementations should verify that incoming values are from valid ranges and that elliptic curve points are contained in the expected group.

The TKLL pseudocode uses some notational conventions that extend the key tree notation from Section 9.1:

- $\text{idx}_{T,p}(v)$ denotes the index of v in $\text{sibling}_T(v)$, with $1 \leq \text{idx}_{T,p}(v) \leq |\text{sibling}_T(v)|$.
- $\text{repped}_{T,p}(v)$ is `TRUE` if v is a k-node in T such that the participant with personal k-node p is represented by another participant in the GKE for $\text{parent}_T(v)$. Otherwise, $\text{repped}_{T,p}(v)$ is `FALSE`. In other words, $\text{repped}_{T,p}(v)$ is `TRUE` if and only if $|\text{sibling}_T(v)| \geq 2$ and the participant with personal k-node p is not associated with the "leftmost" u-node in the subtree rooted at v.
- $\text{path}_T(p)$ denotes the sequence of k-nodes on the path from personal k-node p (inclusive) to the root of T (exclusive). This sequence will contain $\text{depth}_T(p)$ entries.
- N denotes $|\text{userset}(T)|$, the total number of participants.
- The "object" V represents a convenient way to store complex state information between function calls; this greatly simplifies the definitions. V behaves like an object in Javascript: when the code references a field V.field for the first time, this value is instantiated as an "empty" version of the appropriate type (based on the context).

The prototype implementation of TKLL uses Curve25519 [Ber06] to implement Simple KLL (as discussed in Section 9.2). The implementation includes two MS-BN instantiations for the key spaces used by $\text{BRAKEM}^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$. The implementation instantiates Sig and SVerif for the $\text{BRAKEM}^{\text{DDL}}$ key space using the Schnorr signature scheme [Sch91] and for the $\text{BRAKEM}^{\text{ZK}}$ key space using EdDSA.

Formal security proofs for TKLL are left as future work. Informally, the security of Simple KLL and DH exchanges when the CDH problem is hard in the ROM ensures that honest participants do not leak the new keys to adversaries. When the Enc cryptosystem is IND-CCA2 secure, the DH private keys are also protected. Active network adversaries are defeated due to the digital signatures applied to all DH and Simple KLL exchanges; this essentially follows from the security of the Katz-Yung compiler. The only attacks that can be performed by malicious insiders are equivocation-like attacks that result in honest participants reaching different states. However, the multi-signature will only be valid if all honest participants agree about the final state, including public keys that were established for k-nodes outside of their key sets. This mechanism prevents insider attacks against any k-node that contains an honest user in its user set.

Algorithm 9.5 THE TKLL PROTOCOL. (Ref: 311)

Subroutine | $\text{TKLL}(\overrightarrow{ID}, \vec{L}, T, sk) \rightarrow (P^*, Q^*, I^*, \Pi^*, \sigma)$

Let V be an empty object that can store GKE states.
Let p be the personal key in T for the caller (U_1).
$(V, r_1, R_1, t_1) \leftarrow \text{TKLL.Prep}_1(T, p, V)$.
Call $\text{TKLL.Send}_1(T, p, V, t_1)$.
$(V, \vec{t}) \leftarrow \text{TKLL.Receive}_1(T, p, V, t_1)$.
$V \leftarrow \text{TKLL.Prep}_2(T, p, V)$.
Call $\text{TKLL.Send}_2(\overrightarrow{ID}, L_1, T, p, V, sk)$.
$V \leftarrow \text{TKLL.Receive}_2(T, p, V, \vec{L})$.
$(V, P^*, Q^*, I^*) \leftarrow \text{TKLL.Prep}_3(T, p, V)$.
Call $\text{TKLL.Send}_3(T, p, P^*, I^*, R_1)$.
$(P^*, I^*, \vec{R}) \leftarrow \text{TKLL.Receive}_3(T, \vec{t}, P^*, I^*, R_1)$.
$(\Pi^*, \sigma, s_1) \leftarrow \text{TKLL.Prep}_4(\overrightarrow{ID}, \vec{L}, T, \vec{t}, \vec{R}, r_1, sk, P^*, Q^*, I^*)$.
Call $\text{TKLL.Send}_4(T, \Pi^*, s_1)$.
$(\Pi^*, \sigma) \leftarrow \text{TKLL.Receive}_4(\overrightarrow{ID}, \vec{L}, T, \vec{R}, P^*, \Pi^*, \sigma, s_1)$.
return $(P^*, Q^*, I^*, \Pi^*, \sigma)$.

Algorithm 9.6 TKLL ROUND 1 PREPARATION. The participant chooses a DH key pair and prepares the multi-signature commitment. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Prep}_1(T, p, V) \rightarrow (V, r_1, R_1, t_1)$

for $(v \in \text{path}_T(p))$ {
  if $(\text{repped}_{T,p}(v))$ { **break** }
  if $(|\text{sibling}_T(v)| < 2)$ { **continue** }
  $i \leftarrow \text{idx}_{T,p}(v)$.
  $\left(V.l^{(p)}, V.r^{(v)}, V.x_i^{(v)}, V.X_i^{(v)}\right) \leftarrow \text{Simple-KLL}_1(i, |\text{children}_T(v)|)$.
}
$(r_1, R_1, t_1) \leftarrow \text{MSBN.Sign}_1()$.
return (V, r_1, R_1, t_1).

Algorithm 9.7 TKLL ROUND 1 OUTGOING TRANSMISSIONS. The participant sends compressed DH public keys for each GKE and a hash of its multi-signature commitment. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Send}_1(T, p, V, t_1)$

for $(v \in \text{path}_T(p))$ {
  if $(\text{repped}_{T,p}(v))$ { **break** }
  if $(|\text{sibling}_T(v)| < 2)$ { **continue** }
  $i \leftarrow \text{idx}_{T,p}(v)$.
  Broadcast $\text{Compress}_E\left(V.X_i^{(v)}\right)$ to others in $\text{userset}_T(\text{parent}_T(v))$.
}
if $(N > 1)$ {
  Broadcast t_1 to others in $\text{userset}(T)$.
}

Algorithm 9.8 TKLL ROUND 1 INCOMING TRANSMISSIONS. The participant decompresses the incoming DH public keys and receives the hashes of the multi-signature commitments. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Receive}_1(T, p, V, t_1) \rightarrow (V, \vec{t})$

for $(v \in \text{path}_T(p))$ {
  if $(|\text{sibling}_T(v)| < 2)$ { **continue** }
  for $(v' \in \text{sibling}_T(v)$ with sibling index $i)$ {
    if $(i = \text{idx}_{T,p}(v)$ **&& !**$\text{repped}_{T,p}(v))$ { **continue** }
    Receive x from representative for v'.
    $V.X_i^{(v)} \leftarrow \text{Decompress}_E(x)$.
  }
}
for $(2 \leq i \leq N)$ {
  Receive t_i from U_i.
}
Let $\vec{t} = (t_1, \ldots, t_N)$.
return $(V, \vec{t})$.

Algorithm 9.9 TKLL ROUND 2 PREPARATION. The participant computes the "left" and "right" Simple KLL DH shared secrets and the XOR values for each GKE. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Prep}_2(T, p, V) \to V$

for $(v \in \text{path}_T(p))$ {
  if $(\text{repped}_{T,p}(v))$ { **break** }
  if $(|\text{sibling}_T(v)| < 2)$ { **continue** }
  $i \leftarrow \text{idx}_{T,p}(v)$.
  $l \leftarrow V.l^{(v)}$.
  $r \leftarrow V.r^{(v)}$.
  $\left(V.t_{l,i}^{(v)}, V.t_{i,r}^{(v)}, V.Y_i^{(v)}\right) \leftarrow \text{Simple-KLL}_2\left(l, i, r, V.x_i^{(v)}, V.X_l^{(v)}, V.X_r^{(v)}\right)$.
}
return V.

Algorithm 9.10 TKLL ROUND 2 OUTGOING TRANSMISSIONS. The participant sends its Simple KLL XOR values. The participant encrypts and sends the DH private keys for each GKE in which it is a representative. It then signs all outgoing Simple KLL transmissions. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Send}_2(\overrightarrow{ID}, L_1, T, p, V, sk)$

$\kappa \leftarrow \perp$.
for $(v \in \text{path}_T(p))$ {
  if $(\text{repped}_{T,p}(v))$ { **break** }
  if $(|\text{sibling}_T(v)| < 2)$ { **continue** }
  $i \leftarrow \text{idx}_{T,p}(v)$.
  if $(|\text{sibling}_T(v)| > 2)$ {
    Compute $\sigma_i^{(v)} \leftarrow \text{Sig}\Big(L_1, sk, ID_1 \| V.X_i^{(v)} \| V.Y_i^{(v)}\Big)$.
    Broadcast $V.Y_i^{(v)} \| \sigma_i^{(v)}$ to others in $\text{userset}_T(\text{parent}_T(v))$.
  } **else** {
    Compute $\sigma_i^{(v)} \leftarrow \text{Sig}\Big(L_1, sk, ID_1 \| V.X_i^{(v)}\Big)$.
    Broadcast $\sigma_i^{(v)}$ to others in $\text{userset}_T(\text{parent}_T(v))$.
  }
  if $(v \neq p$ **&&** $\kappa \neq \perp)$ {
    Compute $c \leftarrow \text{Enc}\Big(\text{KDF}_1(\kappa), V.x_i^{(v)}\Big)$.
    Broadcast c to others in $\text{userset}_T(v)$.
  }
  $\kappa \leftarrow V.t_{1,2}^{(v)}$.
}

Algorithm 9.11 TKLL ROUND 2 INCOMING TRANSMISSIONS. The participant receives the Simple KLL XOR values, the DH private key ciphertexts sent by the participant's representatives, and the signatures for Simple KLL transmissions. (Refs: 311 and 313)

Subroutine | TKLL.Receive$_2(T, p, V, \vec{L}) \rightarrow V$

for ($v \in \text{path}_T(p)$) {
  if ($|\text{sibling}_T(v)| < 2$) { **continue** }
  for ($v' \in \text{sibling}_T(v)$ with sibling index i) {
    if ($i = \text{idx}_{T,p}(v)$ **&& !**$\text{repped}_{T,p}(v)$) { **continue** }
    Let j be the index in $\vec{L}$ for the representative for v'.
    if ($|\text{sibling}_T(v)| > 2$) {
      Receive $V.Y_j^{(v)} \| \sigma_j^{(v)}$ from U_j.
      if (**!**SVerif$(L_j, ID_j \| V.X_j^{(v)} \| V.Y_j^{(v)}, \sigma_j^{(v)})$) { Abort }
    } **else** {
      Receive $\sigma_j^{(v)}$ from U_j.
      if (**!**SVerif$(L_j, ID_j \| V.X_j^{(v)}, \sigma_j^{(v)})$) { Abort }
    }
  }
  if ($\text{repped}_{T,p}(v)$) {
    Receive $V.c^{(v)}$ from our representative for v.
  }
}
return V.

Algorithm 9.12 TKLL ROUND 3 PREPARATION. The participant derives the final key pairs for k-nodes on its path and computes an identity. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Prep}_3(T, p, V) \to (V, P^*, Q^*, I^*)$

$z \leftarrow \perp$. $\kappa \leftarrow \perp$.
for $(v \in \text{path}_T(p))$ {
  Let $n = |\text{sibling}_T(v)|$.
  if $(n < 2)$ {
    if $(z = \perp)$ {
      Generate a random group key preimage and store it in $V.z^{(v)}$.
    } **else** {
      $V.z^{(v)} \leftarrow \text{KDF}_2(z)$.
    }
    $z \leftarrow V.z^{(v)}$.
    continue.
  }
  $i \leftarrow \text{idx}_{T,p}(v)$.
  if $(i > 1)$ { Let $l = i - 1$ } **else** { Let $l = n$ }
  if $(i < n)$ { Let $r = i + 1$ } **else** { Let $r = 1$ }
  if $(\text{repped}_{T,p}(v))$ {
    $V.x_i^{(v)} \leftarrow \text{Dec}\big(\text{KDF}_1(\kappa), V.c^{(v)}\big)$.
    if ($V.x_i^{(v)}$ does not correspond to $V.X_i^{(v)}$) { Abort }
    $\Big(V.t_{l,i}^{(v)}, V.t_{i,r}^{(v)}, Y'\Big) \leftarrow \text{Simple-KLL}_2\Big(l, i, r, V.x_i^{(v)}, V.X_l^{(v)}, V.X_r^{(v)}\Big)$.
    if $(Y' \neq V.Y_i^{(v)})$ { Abort }
  }
  Let $\vec{Y} = \Big(V.Y_1^{(v)}, \ldots, V.Y_n^{(v)}\Big)$.
  $V.z^{(v)} \leftarrow \text{Simple-KLL}_3\Big(l, i, n, \vec{Y}, V.t_{l,i}^{(v)}, V.t_{i,r}^{(v)}\Big)$.
  if $(V.z^{(v)} = \perp)$ { Abort }
  $z \leftarrow V.z^{(v)}$.
  $\kappa \leftarrow V.t_{1,2}^{(v)}$.
}
for $(v \in \text{path}_T(p))$ {
  $(pk, sk) \leftarrow \text{KeyGen}\big(H\big(V.z^{(v)}\big)\big)$.
  Add mapping $\text{parent}_T(v) \to pk$ to P^* and $\text{parent}_T(v) \to sk$ to Q^*.
}
Add mapping $U_1 \to \text{Compute-ID}(P^*, Q^*)$ to I^*.
return (V, P^*, Q^*, I^*).

Algorithm 9.13 TKLL ROUND 3 OUTGOING TRANSMISSIONS. The participant distributes the public keys for k-nodes that it represents, then broadcasts its identity and multi-signature commitment.
(Refs: 311 and 313)

Subroutine | TKLL.Send$_3(T, p, P^*, I^*, R_1)$

for ($v \in \text{path}_T(p)$) {
  if ($\text{parent}_T(v)$ is the root of T) { **continue** }
  if ($\text{repped}_{T,p}(v)$) { **continue** }
  if ($\text{idx}_{T,p}(v) \neq 1$) { **continue** }
  Locate $\text{parent}_T(v) \longrightarrow pk$ in P^*.
  Broadcast mapping $\text{parent}_T(v) \longrightarrow pk$ to others in $\text{userset}(T) \setminus \text{userset}_T(\text{parent}_T(v))$.
}
Locate $U_1 \longrightarrow I$ in I^*.
Broadcast $I \| R_1$ to others in $\text{userset}(T)$.

Algorithm 9.14 TKLL ROUND 3 INCOMING TRANSMISSIONS. The participant receives the public keys for k-nodes outside its path, the other participants' identities, and the multi-signature commitments.
(Refs: 311 and 313)

Subroutine | TKLL.Receive$_3(T, \vec{t}, P^*, I^*, R_1) \longrightarrow (P^*, I^*, \vec{R})$

for (k-node v in T such that $U_1 \notin \text{userset}_T(v)$) {
  Let U_i be the participant with the "leftmost" u-node in the subtree rooted at v.
  Receive mapping $v \longrightarrow pk$ from U_i.
  Add mapping $v \longrightarrow pk$ to P^*.
}
for ($2 \leq i \leq N$) {
  Receive $I_i \| R_i$ from U_i.
  Add mapping $U_i \longrightarrow I_i$ to I^*.
  if ($H_0(R_i) \neq t_i$) { Abort }
}
Let $\vec{R} = (R_1, \ldots, R_N)$.
return $(P^*, I^*, \vec{R})$.

Algorithm 9.15 TKLL ROUND 4 PREPARATION. The participant finalizes the multi-signature to approve the results and produces a NIZKPK that authenticates its identity. (Refs: 311 and 313)

Subroutine | $\text{TKLL.Prep}_4(\overrightarrow{ID}, \vec{L}, T, \vec{t}, \vec{R}, r_1, sk, P^*, Q^*, I^*) \rightarrow (\Pi^*, \sigma, s_1)$

Let m be a unique encoding of $\left(\overrightarrow{ID}, \vec{L}, T, P^*\right)$.
Add $U_1 \rightarrow$ Prove-ID(I^*, P^*, Q^*) to Π^*.
if $(N < 2)$ {
    $\sigma \leftarrow \text{Sig}(L_1, sk, m)$.
    $s_1 \leftarrow \perp$.
} **else** {
    $\sigma \leftarrow \perp$.
    $s_1 \leftarrow \text{MSBN.Sign}_3\left(\vec{L}, \vec{t}, \vec{R}, r_1, sk, m\right)$.
}
return (Π^*, σ, s_1).

Algorithm 9.16 TKLL ROUND 4 OUTGOING TRANSMISSIONS. The participant broadcasts its multi-signature contribution and its proof of identity.
(Refs: 311 and 313)

Subroutine | $\text{TKLL.Send}_4(T, \Pi^*, s_1)$

if $(N \geq 2)$ {
    Locate $U_1 \rightarrow \pi_1$ in Π^*.
    Broadcast $s_1 \| \pi_1$ to others in userset(T).
}

Algorithm 9.17 TKLL ROUND 4 INCOMING TRANSMISSIONS. The participant receives the other participants' multi-signature contributions and identity proofs. It derives and verifies the final multi-signature. (Refs: 311 and 313)

Subroutine | TKLL.Receive$_4(\overrightarrow{ID}, \vec{L}, T, \vec{R}, P^*, \Pi^*, \sigma, s_1) \rightarrow (\Pi^*, \sigma)$

for $(2 \leq i \leq N)$ {
  Receive $s_i \| \pi_i$ from U_i.
  Add mapping $U_i \rightarrow \pi_i$ to Π^*.
}
if $(N \geq 2)$ {
  Let $\vec{s} = (s_1, \ldots, s_N)$.
  $\sigma \leftarrow$ MSBN.Sign$_4(\vec{L}, \vec{R}, \vec{s})$.
  Let m be a unique encoding of $(\overrightarrow{ID}, \vec{L}, T, P^*)$.
  if (MSBN.Vf$(\vec{L}, m, \sigma) \neq 1$) { Abort }
}
return (Π^*, σ).

9.5 Chapter Summary

This chapter introduced TKLL, a new interactive GKE. While GKEs typically establish a single shared key for all participants, TKLL creates multiple key pairs to instantiate a given key tree. This procedure takes only four rounds of communication, no matter how complex the key tree is. Section 9.1 introduced notation for key trees that will be used in subsequent chapters. Section 9.2 introduced a sub-protocol called Simple KLL: this is an unauthenticated two-round GKE that is essentially the BD protocol [BD94] using the "XOR" operator as in KLL [KLL04]. Section 9.3 introduced the unauthenticated version of TKLL and worked through a specific example. TKLL operates by performing a Simple KLL GKE for each k-node in the key tree. For k-nodes closer to the root, the Simple KLL protocol is executed by representatives for each subtree. These representatives efficiently encrypt the private key material for the GKE to the other participants in their subtree. The protocol covers several edge cases in order to operate correctly and efficiently for any key tree shape. Section 9.4 formally defined TKLL with authentication mechanisms incorporated. The full TKLL protocol is secure against active network adversaries and insider attacks. The participants also exchange identities and authenticate themselves as part of the protocol. The identification functions are given by the higher-level protocol in order to support arbitrary identification schemes, including schemes that generate dynamic identities based on partial TKLL results. The output of TKLL is a set of values that can be sent to existing participants in a secure group messaging scheme in order to non-interactively prove that the key tree was established correctly and that all participants have the same view of the final state.

The TKLL protocol can be used to efficiently implement the "mass join" operation in Safehouse. BRAKEM (defined in Chapter 8) and TKLL are the two core sub-protocols required to define Safehouse. With both prerequisites covered, the next chapter defines the Safehouse protocol.

CHAPTER 10 | Safehouse: A Comprehensive Secure Group Messaging Solution

In this chapter: Protocol Evaluation

SAFEHOUSE is a new cryptographic protocol designed to solve the difficult cryptographic problems at the heart of any secure group messaging scheme, allowing the developer to concentrate on the desired high-level functionality.[1] Chapter 7 outlined the objectives of the Safehouse protocol's design and explained how these objectives were derived from usability studies and the current group messaging landscape. This chapter presents the Safehouse protocol, which makes heavy use of BRAKEM (see Chapter 8) and TKLL (see Chapter 9). To achieve its objectives, Safehouse provides several features that enable the construction of protocols for the target applications described in Chapter 7. At its core, Safehouse establishes an internal secret *group key* that is shared by all members of the group. Safehouse provides an interface for the developer to derive application-specific shared keys from the group key. It also optionally provides sender-specific signing keys, an invitation mechanism, authentication of long-term identities with anonymity preservation, and authenticated storage of arbitrary developer-supplied attributes with optional encryption. The developer can choose which features to use based on hard-coded rules in their application, or based on authenticated group attributes. Notably, Safehouse does not impose a protocol for the actual transmission of payload messages; it is up to the developer to use keys derived from the group key and signing keys in any way they wish. This makes Safehouse equally suitable for sending encrypted text messages with strong reliability guarantees and broadcasting time-sensitive multimedia data in interactive video conferences. However, developers are strongly encouraged to use IND-CCA2 cryptosystems to encrypt payload messages in order to detect misbehavior by the server.

Section 10.1 introduces notation and provides an overview of Safehouse's design. Sections 10.2–10.8 discuss algorithms used internally by Safehouse. Sections 10.9.2, 10.9.3, and 10.10 introduce the interface for Safehouse; the functions in these sections are used by the developer to actually implement a secure group messaging protocol. Section 10.11 discusses a modular component of Safehouse that is a target for future algorithmic optimization work. Section 10.12

[1] The latest version of the Safehouse protocol will be published at safehouse.im.

describes how the developer is expected to use the functions exposed by Safehouse's interface. Finally, Section 10.13 presents the results of a performance evaluation in which Safehouse was used to simulate realistic conversation transcripts.

10.1 Safehouse Design Overview

This section provides an overview of the Safehouse protocol design and how it securely provides the functionality described in Chapter 7. Section 10.1.1 defines core terminology and describes how the various parties interact in the Safehouse protocol, Section 10.1.2 summarizes the data stored by participants, and Section 10.1.3 outlines the major subsystems in Safehouse and their responsibilities. The remaining sections discuss the purpose of the data listed in Section 10.1.2 and the subsystems associated with each field.

10.1.1 Agents and Transmissions

The parties that participate in the Safehouse protocol are called *agents*. This excludes the adversary and other outsiders. The participants that are in the group and have access to the group key are called *members*. The semi-trusted *server* that relays transmissions between the members is also an agent. Together, the one or more members and the one server constitute all of the agents.

Each execution of the Safehouse protocol is called a *session*. Sessions are independent. At any point in time, a session is associated with a specific set of members. Members may join and leave a session over the course of its lifetime. If a developer wants to implement a system that supports simultaneous group conversations with different sets of members, they must create a separate Safehouse session for each conversation. Only the members' long-term identities persist between sessions—and only if the developer has chosen to enable long-term authentication.

Each member is identified by a numeric identifier called an *agent ID*. Every member must have a unique agent ID in the context of a session. In particular, the same agent ID may refer to different members that are in the session at different times, as long as no two members share an agent ID simultaneously at any point. Moreover, since each Safehouse session is independent, an agent that is a member in multiple sessions may have a different agent ID in each session. The developer controls how agent IDs are assigned when new members join a session. From Safehouse's perspective, agent IDs are opaque tokens. The intention is that agent IDs can be used to link entities in the Safehouse session with other data in the higher-level protocol.

Each agent stores a *public state* and a *private state* for the Safehouse session. The combination of an agent's public state and private state is called a *group state*. The server's private state is always empty, whereas members' private states contain cryptographic secrets that must not be disclosed. In particular, members' private states contain the group key. Agents do not need to store any historical states.

An agent's group state changes over time, and the group key changes every time that the group state is updated. An old group state is transformed into a new group state by a special data blob called a *commit* (and in no other way).[2] Commits are created by members and consumed by agents. To create a commit, a member begins by locally initializing a *performer context* with the old group state. Within the performer context, the member performs one or more *operations* that modify the group state in specific ways (e.g., issuing invitations or removing members from the group).[3] Once it has performed all of the desired operations, the member *finishes* the performer context; this procedure produces a commit and the corresponding new group state with all of the operations applied. To consume a commit, an agent locally initializes a *receiver context* with the old group state and the commit. The agent then performs one or more operations within the receiver context. Once complete, the agent *applies* the receiver context.[4] If the operations performed in the receiver context exactly match the operations performed in the performer context that produced the commit, then applying the receiver context yields the corresponding new group state with all of the operations applied; otherwise, the receiver context will report an error and completely discard the pending changes to the group state.

The typical data flow in the Safehouse protocol is that a member creates a commit and sends it to the server, the server applies the commit, the server relays the commit to the other members, and then the other members apply the commit. Note that all agents, including the server, apply commits using receiver contexts, but the server cannot create commits—the server cannot initialize a performer context. Commits can only be created by existing members or by new members that are authorized to join the group.[5] Commits are always digitally signed. In

[2] ^ Safehouse commits are analogous to CGKA commits, as defined in Section 4.4.

[3] ^ Safehouse operations are analogous to CGKA proposals, as defined in Section 4.4

[4] ^ The Safehouse receiver context is analogous to the CGKA "process" function, as defined in Section 4.4.

[5] ^ This differs from MLS, which allows designated outsiders to initiate group state updates in certain situations. For example, MLS can be configured to allow servers authenticated with pre-approved signing keys to add or remove group members [BBM+20, §10.1.4]. A developer can achieve a similar result with Safehouse in some situations by making the server collaborate with a member: when the server wants a commit to be performed, it can refuse to relay commits (and possibly payloads) from members until one of them performs the required commit. The developer can introduce a signaling mechanism for these desired commits and configure the client so that it automatically responds to the server's requests. However, it remains true that a Safehouse server can never unilaterally initiate group updates. This approach is limited to situations in which the server can verify that a commit performed its requested operations.

most cases, a commit is signed by an ephemeral key known only to the specific group member that created the commit. In certain situations where anonymity (among the group members) is required, a commit may be signed by an ephemeral key that is shared by all group members. In either case, the signature cryptographically prevents outsiders from producing commits.

Most operations in the Safehouse protocol accept parameters (e.g., the data to add to the group attributes or the expiry date for a new invitation). The notion of performer and receiver contexts is quite powerful because it does not specify the source of these parameters. Moreover, parameter data is not included in commits. The developer is responsible for deciding which members create commits, what operations they perform, what parameters they use, and when this occurs. The developer specifies these rules as an algorithm called the *group policy*. In many applications, the arguments used for the parameters and the timing of operations is derived from transmissions in a higher-level protocol such as XMPP or a proprietary messaging platform. When this is not the case, the developer can choose to encode operation lists and arguments alongside Safehouse commits without issue; an active network adversary cannot modify arguments attached to a commit without causing the receiver contexts to fail to apply the commit, which is equivalent to simply dropping the commit entirely.

The group state is updated when a performer context is finished or when a receiver context is applied. In both cases, the agent performs operations within a context. In this chapter, the operations are defined in terms of a generic *update context*. In practice, the update context is either a performer context or a receiver context. The operation definitions occasionally use conditional statements based on whether the update context is a performer or receiver context. In either case, the agent executing the operation is referred to as the *processing agent*. This method of definition is convenient because the algorithm for an operation is nearly identical for both performers and receivers: the only difference is that data is written to a commit within a performer context, whereas it is read from a commit within a receiver context. In fact, it is beneficial for operations in real Safehouse implementations to use the same code for both contexts. The operations defined in this chapter refer to the member that executed the performer context as U_{perf}, and to the agent executing the update context (regardless of whether it is a performer context or a receiver context) as U_{proc}. Note that a commit is produced by a single member U_{perf}, whereas every agent applying the commit has a different identifier U_{proc}.

Readers may notice that the "commit" terminology is reminiscent of the `git` version control system. This is no accident: group states in Safehouse behave similarly to common `git` workflows.[6] When Safehouse agents begin with the same public state and apply the same sequence of commits, they will always end up with the same public state. However, honest members always

[6] ^ The process is similar to a `git` workflow in which users push commits to an authoritative central server that is configured to only accept fast-forwards.

have different private states; although the private states may include some common private key material, each member always knows at least one private key that is not known by any other entity. The server is responsible for ensuring that all members apply the same commits in the same order. When the server applies a commit to its own public state, it must store the commit and send it to members upon request. This is how Safehouse provides non-interactivity: members can disconnect from the server for a period of time and then receive the sequence of commits that they missed upon reconnecting. Once the commits are applied, the member ends up with the same public state as other up-to-date members. However, the server should not store commits indefinitely, because the maximum length of time that commits are available sets an upper bound on the *forward secrecy period*: if the adversary corrupts a member's private state, they will be able to recover the group keys for all subsequent states by applying the appropriate commits. The forward secrecy period is the maximum amount of time before group keys can no longer be compromised due to the adversary compromising a private state. In practice, the developer configures the forward secrecy period, all members automatically delete their private state when they have not been able to connect to the server within the forward secrecy period, and the server only stores past commits that were sent within the forward secrecy period. Lengthening the forward secrecy period allows members to be disconnected for longer periods of time without having to rejoin the group, but it also allows the adversary to compromise more messages when corrupting an offline member.

The server is "semi-trusted" because it is responsible for storing and forwarding commits, ensuring the application of a consistent sequence of commits, and automatically deleting commits older than the forward secrecy period. If a malicious server refuses to forward commits, then it essentially performs a denial-of-service attack; keys derived from the group key for one group state will not match keys derived from the group key for a different group state, preventing the successful transmission of messages that have been encrypted using a derived key.[7] A similar effect occurs if a malicious server delivers a different sequence of commits to different members, except that Safehouse guarantees that there is no possible recovery in this scenario: it is intentionally impossible to "merge" group states that have diverged due to a misbehaving server. These attacks that "fork" the group can be made more difficult by replacing the "server" with a set of non-colluding servers that implement the same functionality using a Byzantine fault-tolerant protocol: the honest subset of servers will be able to detect misbehaving servers that do not correctly apply commits in the expected order. Implementing this approach is outside the scope of this work. An honest-but-curious server that stores old public states is not helpful to an adversary; the adversary can learn confidential information only by compromising group members that were entitled to access the confidential information.

[7] ^ The messages will be immediately rejected if the developer encrypts payloads using an IND-CCA2 cryptosystem, as suggested. Otherwise, payloads will be malleable and can be altered by an active adversary.

The public state stored by the server is considered to be the authoritative "current" public state for the session. Members are only able to communicate if they have the same public state (and therefore matching group keys in their private states). Since the server has access to the same public state as the members, the developer-provided group policy that determines which operations can be applied and by whom is able to use all elements of the public state as input. When all agents, including the server, are configured with a consistent group policy, the server is able to reject commits from malicious clients that violate the rules. Consequently, protocols using Safehouse can rely on an honest server to reject malicious group updates before they are forwarded to other group members. If a malicious server forwards the invalid commits anyway, the members will still reject them. The ability to drop malicious commits at the server reduces the bandwidth cost of blocking insider attacks.

10.1.2 Group State

The public state for a Safehouse session, *PS*, contains a variety of information stored in 11 named fields. Safehouse ensures that the contents of the public state are authenticated and that all members eventually derive the same public state. The following fields are stored in the public state:

- *PS.H*: a *state hash*, discussed in Section 10.1.3.
- *PS.G*: the *key graph*, discussed in Section 10.1.4.
- *PS.P*: the *public key map*, discussed in Section 10.1.4.
- *PS.X*: the *exhausted key set*, discussed in Section 10.1.5.
- *PS.S*: the *superfluous edges*, discussed in Section 10.1.5.
- *PS.M*: the *developer mode*, discussed in Section 10.1.6.
- *PS.A*: the *public label-value store*,[8] discussed in Section 10.1.7.
- *PS.E*: the *confidential label-value store*, discussed in Section 10.1.7.
- *PS.I*: the *identity table*, discussed in Section 10.1.8.

[8] ^ Data structures of this type are normally described as "key-value tables". Since the word "key" already has many other meanings in the context of Safehouse, the data structures are referred to herein as "label-value stores" containing "label-value pairs" for clarity.

- *PS.U*: the *unblinding ciphertext table*, discussed in Section 10.1.8.
- *PS.L*: the *layaway table*, discussed in Section 10.1.9.

Key trees—a special case of a key graph in which the graph is a tree structure—were described in depth in Section 9.1. Where applicable, this chapter reuses the key tree notation in the context of key graphs when discussing *PS.G*.

The private state for a session is denoted *SS*. Generally speaking, the information contained in a private state is different for each member. The private state contains data in the following structures:

- *SS.K*: the group key.
- *SS.Q*: the *private key map*, discussed in Section 10.1.4.
- *SS.E*: the *label-value secrets*, discussed in Section 10.1.7.
- *SS.U*: the *unblinding secrets*, discussed in Section 10.1.8.

Establishing the group key stored in the private state, *SS.K*, is the main goal of Safehouse. *SS.K* is a secret key shared by all members. It is not directly exposed to developers. Instead, Safehouse exposes a function that uses a KDF to derive an application-specific shared secret key from *SS.K* in the context of a given domain separation string. The developer can use derived keys for any purpose. Typically, a derived key is used to encrypt payload messages.

10.1.3 Operational Layers

The design of Safehouse incorporates several interconnected subsystems. Figure 10.1 depicts these subsystems and the functional dependencies between them. This section provides a brief overview of the subsystems and how they interact.

Section 10.1.1 discussed the notion of *update contexts*. Developers are expected to store the group state (as outlined in Section 10.1.2) for each agent, and to modify the group state using update contexts. The developer-visible operations that are performed in these update contexts are part of the "MLS layer" of the design. While these operations have nothing to do with the IETF MLS protocol discussed in Section 4.3, they are named this way because they effectively provide the same "message security" functionality (in addition to Safehouse's extra features). Each operation performed in an update context alters the group state in some way. Consequently,

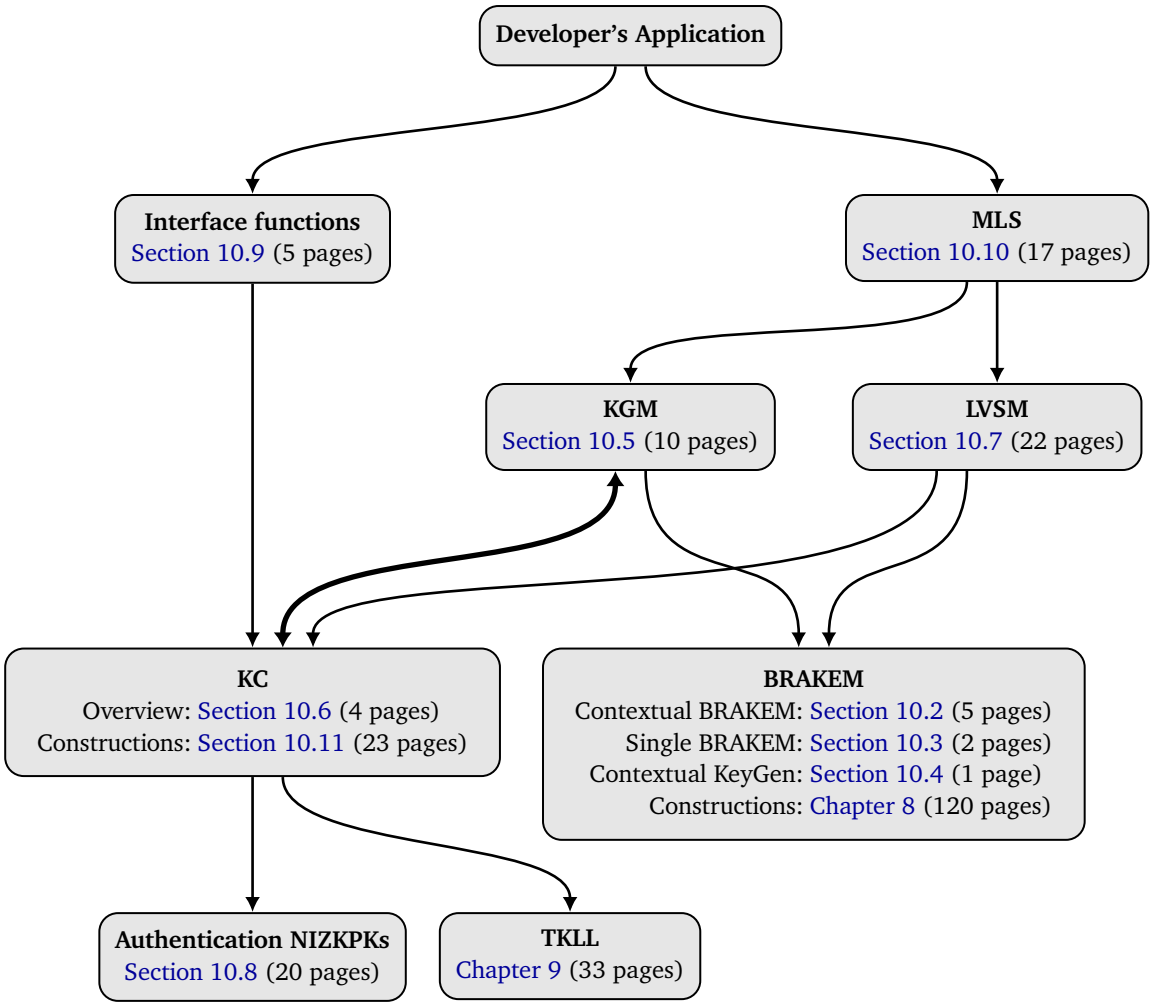


Figure 10.1 SAFEHOUSE SUBSYSTEMS. Directed edges indicate dependencies. For ease of presentation, this figure omits direct dependencies that are already captured by the transitive relationships. The KGM layer depends on the *interface* provided by the KC subsystem, but KC *constructions* depend on operations provided by the KGM layer. (Ref: 329)

update contexts that include more than one operation will implicitly result in an *intermediate group state* between operations. The context keeps track of the current intermediate group state as operations are performed. Operations in the MLS layer are guaranteed to begin and end with a *safe* group state—a state that can be used to encrypt payloads while providing all of the expected security guarantees. Since the only operations that the developer can perform are from the MLS layer, the developer can only interact with safe group states. The set of MLS operations is covered in Section 10.10.

The public API of Safehouse consists of the aforementioned MLS-layer operations (which modify the group state), and several *interface functions* that interact with the group state without modifying it. The interface functions in the API are used to derive keys from the group key, serialize and deserialize group states, and sign messages. Section 10.9 describes the interface functions.

The state hash, $PS.H$, helps to ensure that divergent group states can never be "merged" by the adversary. After each MLS operation, $PS.H$ is set to a hash of the previous value and the commit that caused the group state to change. Consequently, $PS.H$ is derived from the entire history of the group state. All MLS operations partially derive the new group key from the new value of $PS.H$. This ensures that members with diverging group state histories will never be able to communicate, which is an important aspect of Safehouse's insider security guarantees.

The operations in the MLS layer are quite complex. Internally, most of these operations are implemented using developer-inaccessible operations from lower layers of the design. The Key Graph Manipulation (KGM) layer is an internal subsystem consisting of various operations that alter the key graph in the group state. These operations primarily replace keys associated with k-nodes by performing BRAKEM transfers (introduced in Chapter 8) and including the resulting ciphertext and NIZKPKs in the commit. Unlike MLS-layer operations, KGM-layer operations are not required to produce intermediate group states that can securely encrypt payloads. Many KGM-layer operations produce *unsafe* group states. All MLS-layer operations ensure that the group state is safe after performing KGM-layer operations; this procedure is discussed in Section 10.1.5. The set of KGM operations is covered in Section 10.5.

Similarly to the KGM layer, the Label-Value Store Manipulation (LVSM) layer contains internal operations that modify the label-value store in the group state. These operations create, update, or delete values in either the public ($PS.A$) or confidential ($PS.E$) label-value stores, along with their corresponding secrets. In addition to modifying the data stored in the state by the developer, the LVSM layer also includes operations to modify the unblinding ciphertext table ($PS.U$). The LVSM layer is covered in Section 10.7.

Finally, one of the most important components of Safehouse is the Key Control (KC) subsystem. The KC subsystem is responsible for controlling the shape of the key graph in the group state

as members are added to and removed from the session. The shape of the key graph greatly impacts the performance of the scheme: more k-nodes in the graph tends to lead to more efficient BRAKEM transfers (because fewer recipients are needed to provide the required user set), but more keys to update over time and as the session's membership changes. The Safehouse prototype includes several different implementations of the KC API, each causing the key graph to take on a different shape. The best key graph shape to use depends on which MLS operations the developer performs; in general, finding the optimal KC implementation for a given application is a difficult problem that is left to future work. The API provided by the KC subsystem is responsible for six features: initializing the key graph for a new group, keeping track of the structure as KGM operations are performed, efficiently encoding and decoding the group state, identifying the minimal set of k-nodes that contain all members in the union of their user sets, performing an interactive protocol as part of the mass join feature that produces a data blob called a *joining commit*, and implementing an operation that takes a joining commit as a parameter in an update context and adds members to the session. The KGM layer uses the KC API to guide some of its modifications to the group state. In turn, actual implementations of the KC API perform operations provided by the KGM layer. A more detailed description of the KC subsystem is given in Section 10.11.

10.1.4 The Key Graph

The *key graph* for the group, $PS.G$, is defined as a set of u-nodes, a set of k-nodes, and the set of directed edges (which all have a k-node as a destination). This key graph is what provides the core functionality of Safehouse: establishing a shared group key for a group with dynamic membership. At a minimum, for each member in the session, the key graph $PS.G$ contains one u-node for the member and an attached k-node called a *personal key*. The personal key has exactly one incoming edge from a u-node. k-nodes that are not personal keys are called *non-personal keys* or *intermediate k-nodes*. The labels for the u-nodes in the key graph are the agent IDs for the members, while the labels for k-nodes are arbitrarily assigned distinct numbers. The personal key associated with a u-node U is denoted by personal(U). $PS.P$ is a bijection from the k-nodes in $PS.G$ to their corresponding public keys. As a notational convenience, the public key in $PS.P$ associated with a personal key is called the *personal public key*, and the corresponding private key is called the *personal private key*. In general, the key graph is altered by using a BRAKEM scheme to introduce new key pairs as part of a commit. When a new set of members joins the group, they first use TKLL (introduced in Chapter 9) to establish a key tree that is merged into the existing key graph. The group key, $SS.K$, can be updated using new key material that is sent to the whole group using BRAKEM, but $SS.K$ itself is a separate secret key that is not directly represented in the key graph.

SS.*Q* contains all of the private keys in the key graph known to the member. It is a map from a subset of the *k*-nodes in *PS*.*G* to their corresponding private keys. There is exactly one entry in the map stored by member *U* for every *k*-node such that $k \in \text{keyset}_{PS.G}(U)$[9] or $(U, k) \in PS.S$. The contents of *PS*.*S* are discussed in the next section.

During normal operation, Safehouse does not create multiple *k*-nodes associated with the same public key. Malicious insiders can easily cause this to occur, but doing so does not undermine the security properties: the effects are equivalent to the malicious member sending private keys to honest members or to the adversary. In any case, any public keys chosen by malicious insiders (for which the associated private keys may have been improperly shared) are guaranteed to be replaced with fresh key material when the malicious insiders are removed from the group.

For ease of presentation, the algorithms described in this chapter sometimes refer to *k*-nodes based on the public key associated with them in *PS*.*P*. This avoids the need to assign separate names to the keys and the *k*-nodes in contexts where the meaning is always clear. This is purely a notational convenience; actual Safehouse implementations keep track of *k*-nodes based on node identifiers instead of the associated public keys, thereby preventing confusion in the event that a malicious insider forces multiple *k*-nodes to have the same public key.

10.1.5 Superfluous and Exhausted Keys

In some circumstances, it is necessary for a member *U* to perform an operation that replaces a public key *pk* in *PS*.*P* with a new one, even though the associated *k*-node may not be in $\text{keyset}_{PS.G}(U)$. Since the caller of BRAKEM.Encapsulate learns the new private key sk^*, an operation like this results in a private key being learned by a member that should not know it. This fact needs to be stored in the public state; otherwise, a subsequent attempt to evict *U* from the session might fail to replace all private key material known to *U*, potentially allowing a malicious *U* to covertly decrypt new ciphertexts. *PS*.*S* is a bipartite graph that contains *superfluous edges* recording this information. Each edge contained in *PS*.*S* connects a *u*-node to a *k*-node in *PS*.*G*; if the edge connects the *u*-node for *U* to the *k*-node labeled *k*, then it is said that *U* has *superfluous knowledge* of the private key corresponding to *k*. In other words, *U* has knowledge of the private key, but this is undesirable. In this sense, the key graph in *PS*.*G* encodes the *desired* state of private key knowledge, while *PS*.*S* encodes the difference between the desired state and reality. Some operations create superfluous edges, and others eliminate them. The concept of superfluous edges is very similar to the "tainted key" notion used in TTKEM, as discussed in Section 4.5.

[9] This notation was defined in Section 9.1 along with other functions related to key graphs. Recall that $\text{keyset}_{PS.G}(U)$ denotes all *k*-nodes in *PS*.*G* known to *U* (i.e., *k*-nodes which can be reached from *U*'s *u*-node).

As part of the algorithms to perform operations in an update context, members typically perform one or more BRAKEM transfers. The BRAKEM ciphertexts are stored in the commit, which is eventually transmitted over an adversarially controlled network. If the adversary records a BRAKEM ciphertext, the adversary subsequently corrupts a member, and that member has a private key in $SS.Q$ corresponding to one of the BRAKEM recipients, then the adversary will be able to compromise the transferred private key. This type of attack can damage forward secrecy in certain situations; in particular, without additional mitigations, this attack would continue to be applicable for the entire forward secrecy period under most circumstances, because that is the maximum time until all BRAKEM targets must be replaced. Luckily, it is possible to thwart the attack much more quickly, because there is an inexpensive way to non-interactively, publicly verifiably, and irreversibly replace a key pair with a new one, as long as the user set for the associated k-node does not change.

Whenever an operation performs a BRAKEM transfer, the recipient k-nodes are added to the exhausted key set, $PS.X$. An "exhausted key" is a key that needs to be replaced at the end of the commit in order to preserve forward secrecy. The group state is *unsafe* (as defined in Section 10.1.3) whenever $PS.X \neq \emptyset$, so any MLS-layer operation must replace the keys associated with the k-nodes in $PS.X$ before completing; there is no such requirement for KGM-layer operations. Efficiently replacing all of these keys can be done as long as the key space used by the BRAKEM scheme supports the following functions:

- **MixPK**$(pk, pk^*) \rightarrow pk'$ takes two public keys as input, pk and pk^*, and produces a "mixed" public key pk'.
- **MixSK**$(sk, sk^*) \rightarrow sk'$ takes two private keys as input, sk and sk^*, and produces a "mixed" private key sk'. Given public keys pk and pk^* with associated private keys sk and sk^*, if MixPK$(pk, pk^*) \rightarrow pk'$ and MixSK$(sk, sk^*) \rightarrow sk'$, then (pk', sk') is a valid key pair.

These functions are trivial to implement for the BRAKEM constructions defined in Chapter 8. For $\text{BRAKEM}^{\text{DDL}}_{\star}$ (see Section 8.2.1 for notation):

- MixPK$({g_2}^{x_1}, {g_2}^{x_2}) \rightarrow {g_2}^{x_1} \cdot {g_2}^{x_2}$.
- MixSK$(x_1, x_2) \rightarrow x_1 + x_2$.

For $\text{BRAKEM}^{\text{ZK}}$ (see Section 8.5.2 for notation):

- MixPK$(x_1 B, x_2 B) \rightarrow x_1 B + x_2 B$.

- $\text{MixSK}(x_1, x_2) \rightarrow x_1 + x_2$.

At the end of every MLS operation, if $PS.X \neq \emptyset$, then the performer performs a BRAKEM transfer to all members. The newly transferred public key is used to update all k-nodes in $PS.X$ by using MixPK. Members with associated private keys in $SS.Q$ use MixSK with the newly transferred private key to derive the updated private keys. $PS.X$ is then set to $\emptyset$, and all members discard the newly transferred private key. Since the newly transferred private key is never stored by honest members, the adversary cannot recover a private key that was previously used as a BRAKEM target by corrupting a member that has applied the commit. In other words, the exhausted key mechanism is necessary to ensure that the adversary cannot recover private keys that were used only prior to the corrupted member's stored group state.

10.1.6 Configurable Behavior

In order to satisfy all of the requirements discussed in Chapter 7, several aspects of Safehouse's behavior can be configured by the developer. As discussed in Section 10.1.3, the developer is responsible for choosing a BRAKEM construction and a KC implementation. These settings are unchangeable during the lifetime of a Safehouse session. The choice of a BRAKEM construction establishes the key space used by Safehouse, which affects the implementation of multiple subsystems. For example, the BRAKEM construction dictates the implementation of the MixPK and MixSK functions (discussed in Section 10.1.5), the label-value store (discussed in Section 10.1.7), the identification system (discussed in Section 10.1.8), and the authentication system (discussed in Section 10.1.9). Typically, the BRAKEM and KC systems to use are hard-coded as part of the secure messaging tool.

There are four additional Safehouse settings that the developer must specify: the deniability mode, the authentication ephemerality, whether invitations are issued anonymously, and whether post-compromise security is enabled. Safehouse can operate in a non-repudiable mode (where it is possible to cryptographically prove message authorship to third parties), an offline deniable mode, or a strongly deniable mode (with both offline and online deniability).[10] In terms of authentication, members in the group can either be associated with long-term identities (blinded for anonymity preservation), or ephemeral identities that are unlinkable between sessions. Invitations must always be issued by group members, but depending on the configuration, the issuing member may be identified to the other members, or they may be anonymous (other than the fact that they are a member in the session). Post-compromise security can be disabled in order to improve performance. The four configurable settings are specified by the developer and

[10] ^ Offline and online deniability were defined in Section 2.4.

can change over the lifetime of a session. The developer specifies a *group policy* that determines the settings based on *PS*.*M*, an opaque binary string that can be modified through an MLS-layer operation. Typically, either the developer will store some flags in *PS*.*M* that are used by the group policy to determine the settings, or *PS*.*M* will be left empty and the group policy will choose hard-coded settings. Note that in any case, all agents must derive the same deniability and authentication ephemerality settings from the same public state or else the group will be forked. There is one incompatible configuration: when long-term identities are disabled, the deniability is always set to "strongly deniable" mode.[11] To implement the configuration, the developer must implement the following function:

- **SessionConfig**$(M) \rightarrow$ (Deniability, LongTermIDs, InviteAnonymity, PCS): given a mode string M, returns the session configuration. Deniability is one of "non-repudiable", "offline deniable", or "strongly deniable". LongTermIDs is true if long-term identities are enabled, or false if all identities are ephemeral. InviteAnonymity is true if invitations are issued by anonymous members, or false if invitations are issued by identified members. PCS is true if post-compromise security is enabled.

As discussed in Section 10.1.1, in addition to determining the configurable settings, the developer's group policy is also responsible for determining whether a given member may perform a given operation. Whenever an MLS-layer operation is performed in an update context, the group policy is consulted with the intermediate public state, the performing member's agent ID, a description of the operation about to be performed, and the operation's arguments. The group policy returns whether or not the operation is permitted. In addition, the group policy is consulted when a performer context is finished or when a receiver context is applied. In either case, the group policy is given a description of all operations performed in the commit, and it must return whether or not the commit is permitted. These decisions may be made based on hard-coded logic, or based on data stored in the public state. For example, the developer might store a custom moderation mode setting in *PS*.*M*, or member-specific access privileges in *PS*.*A* (discussed in Section 10.1.7). If any of the agents use diverging group policies, then the group will be forked.

10.1.7 Label-Value Stores

PS.*A* is an associative array that contains arbitrary developer-specified label-value pairs. Both the labels and the values are opaque binary strings that are stored unencrypted. Since all agents,

[11] ^ Unauthenticated protocols are strongly deniable by default.

including the server, are able to read the contents of *PS.A*, it may be referenced by the developer-specified group policy that controls what commits may be performed. Like all data in the public state, the integrity of *PS.A* is protected by Safehouse, preventing alteration by adversaries. This is what distinguishes *PS.A* from external data that is simply stored alongside the group state. Typical uses of *PS.A* include storing publicly accessible group descriptions, moderation policies, and data that connects the session to other protocols used by the secure group messaging tool.

PS.E contains an associative array that is like *PS.A*, except that the values are stored as ciphertexts that can only be decrypted by members. The server cannot decrypt the values in *PS.E* and so they cannot be used by the group policy. As in *PS.A*, the labels are not encrypted. New members are able to decrypt the values in *PS.E* immediately upon joining the session. Typical uses of *PS.E* include conversation-specific member attributes, pinned messages, and confidential tokens linking the conversation to other protocols. *PS.E*[`label`] denotes the *encrypted* value identified by "`label`" in the associative array. There are data fields other than the associative array contained in *PS.E* that are used to implement the encryption scheme.

SS.E contains the private keys necessary to decrypt the contents of *PS.E*. It also contains an associative array that maps the labels from *PS.E* to the decrypted plaintexts. *SS.E*[`label`] denotes the *decrypted* value identified by "`label`" in the associative array. There are data fields other than the associative array contained in *SS.E* that are used to implement the encryption scheme.

The contents of the public and confidential label-value stores are modified through the use of MLS-layer operations. Since the values are encrypted in *PS.E*, the group policy cannot dictate the structure of data stored in the values; developers must design their secure messaging tool to tolerate corrupted values with unexpected contents. The operation of *PS.A* and *PS.E*, including details about the encryption mechanism, is covered in greater detail in Section 10.7.

10.1.8 Identification

Safehouse provides a mechanism for a member to identify itself with a long-term public key called its *identity public key*, also known as its *long-term identity*. This mechanism can be enabled or disabled by the developer, as discussed in Section 10.1.6. Members authenticate their identity through a separate mechanism discussed in Section 10.1.9. The identity public keys can be used in a trust establishment scheme (see Section 2.6.1) in order to bind the cryptographic identities to a user's conceptualization of those identities. Members can compare the identity public keys for a session to locally stored trust establishment results.

When long-term identification is enabled, each member provides their identity when first joining the group (or, for the first member in the session, when the group is created). Safehouse

always provides anonymity preservation, so these identities must be hidden from the server and other outsiders. Consequently, the identity public keys are blinded in such a way that only other members can recover them. The blinding process is specific to the selected BRAKEM construction. The following functions are required:

- $\text{Blind}(pk, bpk, bsk) \rightarrow \overline{pk}$ takes as input a public key pk and a key pair (bpk, bsk), then produces a blinded public key $\overline{pk}$. bpk is called the *blinding public key* and bsk is called the *blinding private key*; together they form the *blinding key pair*.
- $\text{Unblind}(\overline{pk}, bpk, bsk) \rightarrow pk$ takes as input a blinded public key $\overline{pk}$ and a blinding key pair (bpk, bsk). If $\text{Blind}(pk, bpk, bsk) \rightarrow \overline{pk}$, then $\text{Unblind}(\overline{pk}, bpk, bsk) \rightarrow pk$. Otherwise, the output of Unblind is a random public key with an undefined probability distribution.

These functions are easy to implement for the BRAKEM constructions defined in Chapter 8. For $\text{BRAKEM}^{\text{DDL}}_{\star}$:

- $\text{Blind}(pk, bpk, bsk) \rightarrow pk^{bsk}$.
- $\text{Unblind}(\overline{pk}, bpk, bsk) \rightarrow \overline{pk}^{bsk^{-1}}$.

For $\text{BRAKEM}^{\text{ZK}}$:

- $\text{Blind}(pk, bpk, bsk) \rightarrow bsk \cdot pk$.
- $\text{Unblind}(\overline{pk}, bpk, bsk) \rightarrow bsk^{-1} \cdot \overline{pk}$.

While neither of these instantiations make use of bpk, it is conceivable that alternative BRAKEM instantiations might benefit from the accessibility of this parameter.

PS.I stores blinded identity public keys for all members, as well as the blinding public keys used to produce them. The same blinding key pairs can be used to blind identities for multiple members. For efficiency, Safehouse tries to alter the group state over time in order to minimize the number of blinding key pairs in use. *PS.I* is a mapping from agent IDs to tuples of the form $(\overline{pk}, bpk)$. Each member's agent ID appears exactly once in *PS.I*. *PS.U* contains a variety of ciphertexts that can be decrypted to recover the blinding private key corresponding to any blinding public key appearing in *PS.I*. These ciphertexts can be decrypted immediately by new members upon joining the session. The implementation of *PS.U* is very similar to the implementation of the confidential label-value store discussed in Section 10.1.7. For this reason, the details are presented in Section 10.7. *SS.U* contains the decrypted private keys recovered from *PS.U*.

To achieve stronger post-compromise security, Safehouse provides a mechanism for an existing member to change their long-term identity. Identity changing commits include a proof that the change was authentic. Like other operations in the MLS layer, the group policy defined by the developer determines when identity changes are permitted, if at all. When long-term identities are disabled by the developer, *PS.I*, *PS.U*, and *SS.U* are all empty.

10.1.9 Invitations and Authentication

There is an inherent conflict between identification, non-interactivity, and anonymity preservation. When members use long-term identities, it is desirable for new members to be able to learn other members' identities immediately upon joining the group. To support non-interactivity, this must be possible without interacting with existing members; in practice, it might be a while before they come back online. Moreover, to achieve anonymity preservation, the adversary must not be able to learn these identities. These requirements lead to the conflict: how can the protocol allow new members to non-interactively learn existing identities, while preventing an honest-but-curious server from doing the same? The only solution is to ensure that new members possess a cryptographic key that is unknown to the adversary. This is the reason that invitations are mandatory in Safehouse: invitations contain private key material that is never shared with the server. This private key material can be used to unblind existing members' identities (as discussed in Section 10.1.8) and to decrypt the contents of the confidential label-value store (as discussed in Section 10.1.7).

Members must be invited to join a Safehouse session. In the first step, an existing member (if permitted by the group policy) performs a commit that creates an invitation. The invitation is a secret blob of data that is returned to the developer after finishing the performer context.[12] The operation that creates the invitation adds a record of its existence to *PS.L*. This record of the invitation is called a *layaway*. The developer (or, more likely, the user) is responsible for securely transmitting the invitation to an entity that wishes to join the group; this out-of-band transmission process is outside the scope of Safehouse. A new member in possession of an invitation can join the session by *claiming* the associated layaway, even if no existing members are available for interactive communication.

Safehouse supports two types of invitations: *bearer* invitations and *individual* invitations. Both types of invitations may optionally have approximate expiry dates that are stored in the layaway; agents will refuse to accept an invitation that has expired, and all members are permitted to perform an operation that deletes expired layaways. Bearer invitations may optionally have a limited number of uses stored in the layaway. If the number of remaining uses is finite, it is

[12] ^ Safehouse invitations are analogous to "welcome messages" in CGKAs, as defined in Section 4.4.

decremented every time that a new member claims the layaway. The layaway is removed as part of the joining process if its remaining uses decreases to zero. Individual invitations are always limited to exactly one use. Moreover, individual invitations are locked to a specific identity public key and may only be used by a new member that provably knows the required identity private key. The required identity is stored in the layaway for individual invitations; this identity is blinded in order to provide anonymity preservation.

While Safehouse mandates the use of invitations to resolve the anonymity preservation conflict, developers can choose to work around the limitation if they wish to implement "open" group conversations. To accomplish this, the developer can have the first member in the session issue a bearer invitation with unlimited uses and no expiry date, and then implement a group policy that prevents the layaway from being deleted. The invitation can then be published in a location that makes it available to all agents who might want to join the "open" group. Note that this approach undermines anonymity preservation and the confidentiality of the confidential label-value store if the invitation is accessible to the adversary. For example, if an honest-but-curious server obtains the invitation, then it could simulate the honest procedure to join the group using the invitation, but not update the group state or distribute the resulting commit. Unlike a scenario in which the adversary uses the invitation to honestly join the group, suppression of the resulting commit by the server allows the compromise to remain invisible to the existing members. In effect, the join operation never actually "happens", but the server nonetheless learns the unblinded identity public keys and the values in the confidential label-value store. Note that since the server does not update the group state, this covert attack does not allow it to decrypt payloads—it only undermines anonymity preservation and the confidential label-value store. This is an inherent limitation that comes from simultaneously supporting confidential data storage in the public state and non-interactive joins that can immediately recover the confidential data: keeping the data confidential requires a cryptographic mechanism of some sort to differentiate legitimate members from adversaries, and if the adversary bypasses the mechanism, then confidentiality can be covertly undermined. The adversary can continue to covertly unblind newly added identity public keys and recover updated confidential values for as long as the layaway associated with the compromised invitation private key remains in $PS.L$; in this approach for implementing "open" groups, the layaway persists indefinitely.

Each invitation contains an *invitation private key* that corresponds to an *invitation public key* in the associated layaway within $PS.L$. Both $PS.E$ and $PS.U$ contain a ciphertext for each layaway. In both cases, the invitation private key can be used to decrypt this ciphertext to recover an *anchor private key* (the anchor private keys used by $PS.E$ and $PS.U$ are distinct). Each value stored in $PS.E$ can be decrypted using the anchor private key for the confidential label-value store. Likewise, every blinding private key used to blind an identity in $PS.I$ can be decrypted using the anchor private key and ciphertexts stored in $PS.U$. This is the mechanism that allows

new members to immediately learn these confidential values. Note that this mechanism requires $PS.E$ and $PS.U$ to be updated every time that a layaway is added to or removed from $PS.L$. The encryption mechanism is described in greater detail in Section 10.7.

$PS.L$ is a set of layaways, where each layaway is a tuple $(iid, ipk, e, bpk, \overline{pk}, u)$ containing the following data:

- iid: a unique numeric identifier for the invitation. In practice, this is used to reference the invitation in operations and from other parts of the group state.
- ipk: the invitation public key.
- e: an approximate expiry date for the invitation, or $\perp$ to indicate that it does not expire based on time.
- bpk: a blinding public key to provide as input to the Blind function (see Section 10.1.8) for individual invitations. For bearer invitations, $bpk = \perp$.
- $\overline{pk}$: a blinded public key for the only member that is permitted to use the invitation. For bearer invitations, $\overline{pk} = \perp$.
- u: an integer indicating how many new members may join the session using the invitation. If $u = \perp$, then the invitation has unlimited remaining uses. For individual invitations, u is always implicitly equal to 1.

An invitation is a tuple $(iid, ipk, isk, bpk, \overline{pk})$, where the values are defined as in the layaway, except that isk is the invitation private key.

Invitations allow new members to immediately learn the other members' identities, but these mechanisms do not provide authentication of the identities. Implementing this functionality non-interactively while offering anonymity preservation would require storing encrypted authentication data in the public state. However, this method conflicts with deniability: the available constructions that provide deniable authentication require members to know the identities of the members that are verifying their credentials. This topic is revisited in Section 10.8. Even when Safehouse is configured to provide non-repudiation (i.e., no deniability), storing encrypted authentication data imposes expensive performance costs. For these reasons, authentication of long-term identities in Safehouse is achieved through a separate, semi-interactive mechanism.

When long-term identities are enabled, new members authenticate to existing members immediately as part of the joining procedure. This authentication is an operation that is performed as part of the same commit that adds the member to the session. This means that a member in

a Safehouse session can always immediately identify and authenticate members that join after it, even if it was offline at the time. In contrast, new members do not immediately receive any authentication for existing members when joining. Instead, existing members authenticate to new members as soon as they can by performing an authentication operation and sending the resulting commit to the server.

Since existing members are not immediately authenticated to new members, there is a period of time after joining a session in which a member may know claimed identities for existing members, but without any proof that these identities are correct. Developers that use long-term identities in their Safehouse sessions must design their secure messaging tool to handle this distinction between "identified" and "authenticated" states, in addition to implementing a trust establishment scheme.

When deniability is active for a Safehouse session, the aforementioned authentication operations (both to existing or new members) may be forged using an appropriate set of private keys. The exact deniability mechanism is discussed in Section 10.8. In order to account for this possibility, the data used to prove a member's identity is referred to as the *identity witness*. This witness may be the identity private key in the honest case, or it may be a set of specific other private keys when forging a transcript as part of a deniability scenario.

The Safehouse deniability mechanism allows authentication operations to be simulated by the forger even without access to an invitation private key, which may have been generated by an online judge. Since the invitation private key is normally used to decrypt the confidential label-value store and unblind long-term identities (if applicable), a forger without an invitation private key cannot recover these values in the normal way. An offline forger will already have access to the data since they forge the entire history of the session. An online forger (i.e., a misinformant) can copy the data from an existing private state under their control.[13] A *state unlocker* is an object that contains the secrets necessary to decrypt the contents of $PS.E$ and unblind the identities in $PS.I$. In the honest case, the state unlocker contains an invitation identifier and an invitation private key. In the forging case when deniability is enabled, the state unlocker contains the already-decrypted values copied from another private state.

[13] ^ A misinformant in the online deniability scenario can instantiate a simulated Safehouse session if the online judge requires them to join a new session, rather than compromising an existing session. Regardless of whether the session is newly created or already existed, the misinformant will have access to the decrypted contents of the confidential label-value store and unblinded long-term identities as part of its private state for one of the session members.

10.2 Contextual BRAKEM

The BRAKEM definition presented in Chapter 8 works well as a standalone primitive that is applicable to protocols outside of secure group messaging. It is a self-contained definition that allows for a game-based proof of security, which is much easier to specify and understand than a composable security proof in a framework like GUC. Safehouse can be entirely implemented using the BRAKEM functions defined in Section 8.1. However, the performance of Safehouse can be significantly improved by defining a lower level interface. Specifically, Safehouse benefits from being able to provide the new key pair to transfer as an input (instead of being generated by the BRAKEM scheme), and being able to sequentially perform transfers to individual sets of recipients, while still aggregating the NIZKPKs for performance. This section defines Contextual BRAKEM (CBRAKEM), which performs the exact same computations as the general BRAKEM construction in Section 8.1.1, except broken down into more granular functions.

A CBRAKEM scheme consists of the following functions:

- **CBRAKEM.KeyGen**$(s) \to (pk, sk)$: Takes as input randomness s, then outputs a public key pk with corresponding private key sk.
- **CBRAKEM.PerformerContext**$() \to ctx$: Creates a new "performer context", ctx, which can perform a sequence of encapsulations.
- **CBRAKEM.ReceiverContext**$(T) \to ctx$: Creates a new "receiver context", ctx, which can perform a sequence of decapsulations or verifications. Takes as input a commit T that was produced by a performer context.
- **CBRAKEM.Encapsulate**$(ctx, R, pk^*, sk^*; s') \to ctx'$: Takes as input a performer context ctx, a set of public keys R, a new key pair (pk^*, sk^*), and randomness s', then outputs a modified context ctx'. If ctx is not a performer context, the function returns $\perp$.
- **CBRAKEM.Decapsulate**$(ctx, j, sk, R, pk^*) \to (ctx', sk^*)$: Takes as input a receiver context ctx, a set of public keys R, an index j identifying a public key in R, a private key sk corresponding to the identified public key, and a new public key pk^*. Outputs either a modified context ctx' and a private key sk^* corresponding to pk^*, or $\perp$.
- **CBRAKEM.Verify**$(ctx, R, pk^*) \to (ctx', b)$: Takes as input a receiver context ctx, a set of public keys R, and a new public key pk^*. Outputs a modified context ctx' and a bit $b \in \{0, 1\}$.

- **CBRAKEM.Finish**$(ctx; s'') \rightarrow T$: Finalizes the transfers made in a performer context ctx using randomness s''. Outputs a commit T that can be used to receive the same sequence of transfers.
- **CBRAKEM.Apply**$(ctx) \rightarrow b$: Finalizes the transfers that were decapsulated or verified in a receiver context ctx. Outputs a bit $b \in \{0, 1\}$ indicating if all of the previously received transfers are valid and no unread transfers remain in the commit.

This version of BRAKEM is similar to the update contexts used by the other operational layers in Safehouse, as discussed in Section 10.1.3. When the performer context is finished, the caller is given a commit T. This opaque blob contains all of the ciphertexts for the encapsulations that were performed, all of the NIZKPKs, and an aggregated Fiat-Shamir challenge for the NIZKPKs. This commit is given as the input to a receiver context. As decapsulations and verifications are performed in the receiver context, the ciphertexts are read from the commit stored in the context. CBRAKEM.Apply returns $b = 1$ if and only if the sequence of transfers in the receiver context match the encapsulations in the performer context. There are a few important differences between CBRAKEM and BRAKEM as defined in Section 8.1:

- In BRAKEM, the Encapsulate, Decapsulate, and Verify functions all take a set of sets of public keys, allowing them to transfer m private keys to m sets of recipients simultaneously. This allows BRAKEM to batch the various NIZKPKs into $m+1$ proofs; $\text{BRAKEM}_\star^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$ use the Π_0 and Π_1 NIZKPKs, respectively, in order to efficiently batch the terms in the proof statements. In CBRAKEM, the functions only transfer one private key to one set of recipients. The same NIZKPKs are still produced, but this occurs during the call to CBRAKEM.Finish instead of during the CBRAKEM.Encapsulate calls. The performer context keeps track of the information necessary to produce these NIZKPKs when the context is finished.
- In BRAKEM, the new key pairs (pk^*, sk^*) are generated within the construction and returned to the caller. In CBRAKEM, the caller provides the key pair as input. This does not alter the actual steps of the algorithm, but it does raise the possibility that sk^* could be leaked if it is mishandled by the caller. Safehouse is carefully designed to preserve the BRAKEM key secrecy property when using CBRAKEM. There is only one place in Safehouse where a new private key sk^* is used for a secondary purpose: the "share key" operation in the KGM layer, presented in Section 10.5.10. This operation is only used in situations where sk^* is the output of a TKLL protocol execution, which also provides key secrecy.
- In CBRAKEM receiver contexts, CBRAKEM.Decapsulate immediately returns new private keys, even though the associated NIZKPKs are not verified until a later call to CBRAKEM.Apply. This means that a malicious prover can cause invalid private keys to be used in subsequent

operations. However, this is not a problem for Safehouse: no matter what operations are performed using a malicious private key, the misbehavior will eventually be identified when CBRAKEM.Apply is called. When CBRAKEM.Apply returns 0 to indicate malicious behavior, the caller will reject the entire commit, discarding any computations that were performed.

A CBRAKEM receiver context completes successfully only if no Decapsulate call returns $\perp$, no Verify call returns $b = 0$, and the Apply call returns $b = 1$. It is the caller's responsibility to abort processing as soon as a Decapsulate or Verify call fails; continuing to use the original or updated context after a failure results in undefined behavior. When Safehouse encounters an error during CBRAKEM decapsulation or verification, it immediately aborts the MLS-layer receiver context and rejects the associated commit.

It is straightforward to transform a BRAKEM construction into a CBRAKEM construction if it follows the design introduced in Section 8.1.1. The context must track the transfers that have occurred, and the NIZKPKs are moved into the Finish or Apply functions. This method can be used to transform $\text{BRAKEM}_{\star}^{\text{DDL}}$, as defined in Section 8.2.8, into $\text{CBRAKEM}_{\star}^{\text{DDL}}$, the equivalent CBRAKEM scheme. The revised scheme operates as follows:

- **$\text{CBRAKEM}_{\star}^{\text{DDL}}$.KeyGen**$(s) \rightarrow (pk, sk)$:

  Choose sk in $\mathbb{Z}_{p_3}$ using randomness in s.
  Compute $pk \leftarrow {g_2}^{sk}$.

- **$\text{CBRAKEM}_{\star}^{\text{DDL}}$.PerformerContext**$() \rightarrow (ctx)$:

  ctx.Type $\leftarrow$ `Performer`.
  Set $ctx.P$ to an empty sequence.
  return ctx.

- **$\text{CBRAKEM}_{\star}^{\text{DDL}}$.ReceiverContext**$(T) \rightarrow (ctx)$:

  ctx.Type $\leftarrow$ `Receiver`.
  $ctx.T \leftarrow T$.
  Set $ctx.V$ to an empty sequence.
  return ctx.

- **$\text{CBRAKEM}_{x}^{\text{DDL}}$.Encapsulate**$(ctx, R, pk^*, sk^*; s') \rightarrow (ctx')$:

  if (ctx.Type $\neq$ `Performer`) **return** $\perp$.
  Derive all subsequently required randomness from s'.

Choose ElGamal secret $r \xleftarrow{\$} \mathbb{Z}_{p_3}$.
Compute $\underline{y} \leftarrow {g_2}^r$.
Compute $\overline{sk^*} \leftarrow {sk^*}^{-1} \pmod{p_2}$.
Compute ElGamal ciphertext $h \leftarrow x^r \times \overline{sk^*} \pmod{p_2}$.
for each $(1 \le j \le |R|)$ {
    Compute ElGamal ciphertext $h_j \leftarrow {R_j}^r \times \overline{sk_i^*} \pmod{p_2}$.
}
Compute $k \leftarrow {g_1}^{sk^*}$.
$C \leftarrow (y, h_1, \ldots, h_{|R|})$.
$ctx' \leftarrow ctx$.
Append $(R, pk^*, sk^*, r, y, h, h_1, \ldots, h_{|R|}, k, C)$ to sequence $ctx'.P$.
return ctx'.

- **CBRAKEM$_x^{\text{DDL}}$.Decapsulate**$(ctx, j, sk, R, pk^*) \to (ctx', sk^*)$:

if $(ctx.\text{Type} \neq \texttt{Receiver})$ **return** $\perp$.
if $(!\ 1 \le j \le |R|)$ **return** $\perp$.
if $(!\ R_j \neq {g_2}^{sk})$ **return** $\perp$. ▷ Guaranteed by insider security; can omit for speed
$ctx' \leftarrow ctx$.
Read $(y, h_1, \ldots, h_{|R|})$ from the start of $ctx.T$
    ↪ and set $ctx'.T$ to the remaining data.
if (reading the data from $ctx.T$ failed due to an invalid transmission) **return** $\perp$.
Compute $sk' \leftarrow {y^{sk}}/{h_j}$.
$sk^* \leftarrow sk' \pmod{p_3}$.
Append $(R, pk^*, y, h_1, \ldots, h_{|R|})$ to sequence $ctx'.V$.
return (ctx', sk^*).

- **CBRAKEM$_\star^{\text{DDL}}$.Verify**$(ctx, R, pk^*) \to (ctx', b)$:

if $(ctx.\text{Type} \neq \texttt{Receiver})$ **return** $\perp$.
$ctx' \leftarrow ctx$.
Read $(y, h_1, \ldots, h_{|R|})$ from the start of $ctx.T$
    ↪ and set $ctx'.T$ to the remaining data.
if (reading the data from $ctx.T$ failed due to an invalid transmission) **return** $(ctx', 0)$.
Append $(R, pk^*, y, h_1, \ldots, h_{|R|})$ to sequence $ctx'.V$.
return $(ctx', 1)$.

- **CBRAKEM$_x^{\text{DDL}}$.Finish**$(ctx; s'') \to T$:

if (ctx.Type $\neq$ `Performer`) **return** $\perp$.
Derive all subsequently required randomness from s''.
Let $m = |ctx.P|$.
simultaneously, merging Fiat-Shamir challenges into c {
  for each $(1 \leq i \leq m)$ {
    Load $(R_i, pk_i^*, sk_i^*, r_i, y_i, h_i, h_{i,1}, \ldots, h_{i,|R_i|}, k_i, C_i)$ from $ctx.P_i$ into scope.
    Prove $\pi_i = \text{BDLEQ}\{r_i : (g_2, y_i) \approx ({}^{R_{i,1}}/_{x}, {}^{h_{i,1}}/_{h_i}) \approx \ldots \approx ({}^{R_{i,|R_i|}}/_{x}, {}^{h_{i,|R_i|}}/_{h_i})\}$.
  }
  Prove $\pi_0 = \Pi_0\{(r_1, \ldots, r_m, x^{r_1}, \ldots, x^{r_m}, sk_1^*, \ldots, sk_m^*) :$
    $x, (k_1, h_1, y_1, pk_1^*) \approx \cdots \approx (k_m, h_m, y_m, pk_m^*)\}$.
}
$T \leftarrow (C_1, \ldots, C_m, c, \pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m)$.
return T.

- **CBRAKEM$_\star^{\text{DDL}}$.Apply**$(ctx) \rightarrow b$:

if (ctx.Type $\neq$ `Receiver`) **return** $\perp$.
Read $c, \pi_0, \pi_1, \ldots, \pi_m, h_1, \ldots, h_m, k_1, \ldots, k_m$ from the start of $ctx.T$
  $\hookrightarrow$ and set T' to the remaining data.
if (reading the data from $ctx.T$ failed due to an invalid transmission) **return** 0.
if (T' is not empty) **return** 0.
Let $m = |ctx.V|$.
simultaneously {
  for each $(1 \leq i \leq m)$ {
    Load $(R_i, pk_i^*, y_i, h_i, h_{i,1}, \ldots, h_{i,|R_i|})$ from $ctx.V_i$ into scope.
    Compute the random oracle inputs for π_i using c from the prover.
  }
  Compute the random oracle inputs for π_0 using c from the prover.
}
Apply the random oracle to the inputs for $\pi_0, \pi_1, \ldots, \pi_m$ to produce c'.
if $(c' \neq c)$ **return** 0.
return 1.

Note that CBRAKEM$_\star^{\text{DDL}}$ also performs the standard optimization of combining the Fiat-Shamir challenges for all NIZKPKs into a single challenge c. This means that all of the NIZKPKs for $\pi_0, \pi_1, \ldots, \pi_m$ are executed simultaneously until they reach the call to the programmable random oracle. Once the prover has reached this state for NIZKPKs, the inputs to the random oracles are concatenated in an unambiguous order before being fed to a single batched random oracle

call. The resulting challenge c is used to simultaneously complete all of the NIZKPKs. Due to the combined challenge, the proofs $\pi_0, \pi_1, \ldots, \pi_m$ do not need to include duplicated copies of c. Note also that the ciphertexts $C_1, \ldots, C_m$ and the combined challenge c must be provided in the commit T *before* any of the proofs $\pi_0, \pi_1, \ldots, pi_m$ in order for the receiver to decapsulate the private keys and verify the proofs.

The alternative BRAKEM construction based on zk-SNARKs presented in Section 8.5.5, $\text{BRAKEM}^{\text{ZK}}$, can be transformed into a CBRAKEM scheme using the same approach that was used for $\text{CBRAKEM}^{\text{DDL}}_{\star}$ above. The resulting scheme is referred to as $\text{CBRAKEM}^{\text{ZK}}$. The contents of the context and how the functions manipulate it are the same as in $\text{CBRAKEM}^{\text{DDL}}_{\star}$; the only difference is that the ciphertexts and proofs are produced as in $\text{BRAKEM}^{\text{ZK}}$. Since the transformation is straightforward, the complete definition of $\text{CBRAKEM}^{\text{ZK}}$ is omitted.

When deploying Safehouse, the developer must decide which CBRAKEM construction to use. This could be $\text{CBRAKEM}^{\text{DDL}}_{\star}$, $\text{CBRAKEM}^{\text{ZK}}$, or a different construction. In any case, the operations in this chapter simply refer to the generic names for the CBRAKEM functions; this is implicitly understood to refer to the construction chosen by the developer.

10.3 Single BRAKEM

It is often necessary to use BRAKEM to transfer a specific private key to a single set of receivers. This functionality is required by the LVSM and KC layers. In these situations, batching together the encapsulation of multiple private keys is not possible. Unfortunately, the general BRAKEM interface defined in Section 8.1 does not support private keys generated by the caller, and the CBRAKEM scheme described in Section 10.2 supports additional features that harm presentational clarity. As a notational convenience, this section defines three wrapper functions that produce and consume a single BRAKEM encapsulation with a key pair specified by the caller. Together, these functions are referred to as a Single BRAKEM (SBRAKEM) scheme. The encapsulation, decapsulation, and verification functions are shown in Algorithms 10.1, 10.2, and 10.3, respectively.

Algorithm 10.1 BRAKEM ENCAPSULATION FOR A SINGLE SET. (Ref: 348)

Subroutine | **SBRAKEM.Encapsulate**$(R, pk^*, sk^*; s) \longrightarrow T$

Derive randomness s' and s'' from s.
$ctx \leftarrow$ CBRAKEM.PerformerContext().
$ctx' \leftarrow$ CBRAKEM.Encapsulate$(ctx, R, pk^*, sk^*; s')$.
$T \leftarrow$ CBRAKEM.Finish$(ctx'; s'')$.
return T.

Algorithm 10.2 BRAKEM DECAPSULATION FOR A SINGLE SET. (Ref: 348)

Subroutine | **SBRAKEM.Decapsulate**$(R, j, sk, pk^*, T) \longrightarrow sk^*$

$ctx \leftarrow$ CBRAKEM.ReceiverContext(T).
$(ctx', sk^*) \leftarrow$ CBRAKEM.Decapsulate(ctx, j, sk, R, pk^*).
if $(sk^* = \perp)$ { **return** $\perp$. }
if (CBRAKEM.Apply$(ctx') \neq 1$) { **return** $\perp$. }
return sk^*.

Algorithm 10.3 BRAKEM VERIFICATION FOR A SINGLE SET. (Ref: 348)

Subroutine | **SBRAKEM.Verify**$(R, pk^*, T) \longrightarrow b$

$ctx \leftarrow$ CBRAKEM.ReceiverContext(T).
$(ctx', b) \leftarrow$ CBRAKEM.Verify(ctx, R, pk^*).
if $(b \neq 1)$ { **return** 0. }
if (CBRAKEM.Apply$(ctx') \neq 1$) { **return** 0. }
return 1.

10.4 Contextual Key Generation

In several circumstances, it is useful to generate a new key pair in a performer context while keeping the private key secret (i.e., not transferring it to other parties using a scheme like CBRAKEM). This operation is denoted **CKeyGen** ("contextual key generation"). CKeyGen operates as follows:

Output: pk^*, a new public key; and sk^*, either the private key corresponding to pk^* or $\perp$.

---

if (performer context) {
    $(pk^*, sk^*) \leftarrow$ CBRAKEM.KeyGen().
    Add pk^* to the commit.
} **else** {
    Read pk^* from the commit.
    $sk^* \leftarrow \perp$.
}
return pk^* and sk^*.

10.5 Key Graph Manipulation

The KGM layer provides fundamental operations that safely alter the state of the key graph, as discussed in Section 10.1.3. These operations primarily involve modifying the key graph $PS.G$, the superfluous edges $PS.S$, the exhausted key set $PS.X$, and the associated private keys (see Section 10.1.2 for an overview of the group state). KGM operations are implicitly given access to the commit: performer contexts may append data to the commit being constructed, and receiver contexts may read and remove a prefix from the commit being consumed.

The algorithms for operations often involve performing helper operations as subprotocols. The pseudocode in this chapter uses an alternative notation to denote the execution of a helper operation: the named parameters and named return values are listed on individual lines prefixed with the "▸" symbol. This convention helps to distinguish operations from typical function calls; helper operations are also given implicit access to the commit and the intermediate group state, just like the calling operation.

10.5.1 Key Transfer

The **KGM.KeyTransfer** operation uses CBRAKEM to send a private key to a set of recipients. This is a low-level operation that is used only as a subroutine by other operations in the KGM layer. For this reason, the operation does not alter the group state; it provides its output directly to the caller. The new public key pk^* is provided as a parameter. The performer also provides the corresponding private key sk^* as a parameter, whereas the receiver does not provide the private key. If U_{proc} is the performer or one of the intended recipients, then the operation outputs the new private key sk^*.

Parameters: R, a set of k-nodes; pk^*, a public key; sk^*, either the private key corresponding to pk^* or $\perp$; and *bctx*, a CBRAKEM update context.
Output: sk^*, either the private key corresponding to pk^* or $\perp$; and *bctx*, the updated CBRAKEM context.

Form K, the set of public keys from $PS.P$ corresponding to the k-nodes in R.
if (performer context) {
  if ($sk^* = \perp$) { Abort. }
  $bctx \leftarrow$ CBRAKEM.Encapsulate($bctx, R, pk^*, sk^*$).
} **else** {
  if ($\exists r \in R$ such that $r \rightarrow sk$ is in $SS.Q$) {
    Let j identify r in R.
    ($bctx, sk^*$) $\leftarrow$ CBRAKEM.Decapsulate($bctx, j, sk, R, pk^*$).
    if ($sk^* = \perp$) { Abort. }
  } **else** {
    ($bctx, v$) $\leftarrow$ CBRAKEM.Verify($bctx, R, pk^*$).
    if ($v \neq 1$) { Abort. }
    $sk^* \leftarrow \perp$.
  }
}
for each ($k \in R$) {
  if ($k \notin PS.X$) { Add k to $PS.X$. }
}
return sk^*.

10.5.2 New Key Transfer

The **KGM.NewKeyTransfer** operation is a small wrapper that generates a new key pair using CKeyGen (see Section 10.4) and uses KGM.KeyTransfer to transfer the private key. Like KGM.KeyTransfer, it is used only as a subroutine by other operations in the KGM layer.

Parameters: R, a set of k-nodes; and *bctx*, a CBRAKEM update context.
Output: pk^*, a new public key; sk^*, either the private key corresponding to pk^* or $\perp$; and *bctx*, the updated CBRAKEM context.

Execute CKeyGen with:
- Output pk^*: store in pk^*
- Output sk^*: store in sk^*.

Execute KGM.KeyTransfer with:
- Parameter R: set to R
- Parameter pk^*: set to pk^*
- Parameter sk^*: set to sk^*
- Parameter *bctx*: set to *bctx*
- Output sk^*: store in sk^*
- Output *bctx*: store in *bctx*.

return pk^*, sk^*, and *bctx*.

10.5.3 Discard Member

The **KGM.DiscardMember** operation simply erases an existing member from $PS.G$. A member can only be removed if it does not know the private keys for any k-nodes except for its personal key.

Parameters: U, the agent ID for the member to discard.

---

if $(\text{keyset}_{PS.G}(U) \neq \{\text{personal}(U)\})$ { Abort. }
if ($PS.S$ contains a mapping from U) { Abort. }
if $(\text{personal}(U) \in PS.X)$ { Remove personal(U) from $PS.X$. }
Remove personal(U) from $PS.G$.
Remove u-node U from $PS.G$.
if $(U_{\text{proc}} = U)$ { Set SS to $\perp$. }

10.5.4 Merge Keys

The **KGM.MergeKeys** operation "merges" two source keys into a new destination key in $PS.G$. The user set of the destination key is the union of the user sets for the two source keys. In other words, any member with knowledge of at least one source private key learns the destination private key.

Parameters: s_1 and s_2, the two source k-nodes; and *bctx*, a CBRAKEM update context.
Output: *bctx*, the updated CBRAKEM context.

Execute KGM.NewKeyTransfer with:
- Parameter R: set to $\{s_1, s_2\}$
- Parameter *bctx*: set to *bctx*
- Output pk^*: store in pk^*
- Output sk^*: store in sk^*
- Output *bctx*: store in *bctx*.

Add a new k-node d to $PS.G$.
Add edges from s_1 to d and from s_2 to d in $PS.G$.
if $(sk^* \neq \perp)$ { Add mapping $d \rightarrow sk^*$ to $SS.Q$. }
if $(d \notin \text{keyset}_{PS.G}(U_{\text{perf}}))$ { Add edge $U_{\text{perf}} \rightarrow d$ to $PS.S$. }
for each $(U \rightarrow s_1$ in $PS.S)$ {
  if $(d \notin \text{keyset}_{PS.G}(U))$ { Add the mapping $U \rightarrow d$ to $PS.S$. }
}
for each $(U \rightarrow s_2$ in $PS.S)$ {
  if $(d \notin \text{keyset}_{PS.G}(U))$ { Add the mapping $U \rightarrow d$ to $PS.S$. }
}
return *bctx*.

10.5.5 Add Edge

The **KGM.AddEdge** operation adds an edge to $PS.G$. This operation can only be performed if it does not alter the key sets of any u-node in the graph. Equivalently, the user set of the destination k-node for the new edge is not altered.

Parameters: s, a source k-node; and d, a destination k-node.

---

if (an edge $s \rightarrow d$ already exists in $PS.G$) { Abort. }

if ($\text{userset}_{PS.G}(d) \not\subseteq \text{userset}_{PS.G}(s)$) { Abort. }
Add an edge from s to d in $PS.G$.

10.5.6 Discard Key

The **KGM.DiscardKey** operation removes a given k-node from $PS.G$. Personal keys cannot be discarded. Edges are added to the key graph so that the semantic relations between other k-nodes are not altered by the operation.

Parameters: k, a k-node to discard.

---

if (k is a personal key) { Abort. }
for each (k-node s such that $PS.G$ contains an edge from s to k) {
  for each (k-node d such that $PS.G$ contains an edge from k to d) {
    if (the only paths from s to d in $PS.G$ pass through k) {
      Add an edge from s to d in $PS.G$.
    }
  }
}
Remove k and all edges involving k from $PS.G$.
If present, remove any edges to k in $PS.S$.
If present, remove k from $PS.X$.
If present, remove the mapping from k in $SS.Q$.

The test in KGM.DiscardKey to see if there are alternate paths from s to d in $PS.G$ is inexpensive in practice. The key graph is always a DAG, and it is often additionally a tree (for which the test is trivial). Moreover, in typical deployments, the maximum path length in the key graph is less than 5.[14] Consequently, it is reasonable to implement the test as a simple depth-first search through $PS.G$.

10.5.7 Evict

The **KGM.Evict** operation replaces all non-personal keys known by a set of members with new keys. The key graph is then updated to reflect the new situation: each one of the specified

[14] The performance of Safehouse in typical configurations with predictable key graph structures is explored in Section 10.13.

members will not have knowledge of any private keys except for its own personal key. The member that performs this operation cannot be in the set of members to evict.[15] The operation uses a variety of techniques to replace keys based on the shape of the key graph. In special cases where a k-node is left with a single child k-node with the same user set, the k-nodes can be merged together. However, since this may be undesirable for certain KC implementations, this option is only used at the discretion of the KC subsystem. The function **KC.MayMerge**$(G, s, k) \to b$ returns $b = 1$ if it is acceptable to merge k-nodes s and k (see Section 10.6). In the general case, keys are replaced using the CBRAKEM scheme.

Parameters: U, a set of agent IDs specifying members to evict; and *bctx*, a CBRAKEM update context.
Output: *bctx*, the updated CBRAKEM context.

---

if $(U_{\text{perf}} \in U)$ { Abort. }
Let $K' = \{k \mid \exists u \in U(k \in \text{keyset}_{PS.G}(u) \wedge k \neq \text{personal}(u))\}$.
Let $K'' = \{k \mid \exists u \in U(PS.S \text{ maps } u \to k)\}$.
$K \leftarrow K' \cup K''$.
Compute S, a topological sort of K.
for each ($k \in S$ in topological order) {
  $c \leftarrow \emptyset$.
  for each (k-node s such that an edge from s to k is in $PS.G$) {
    if $(\exists u \in U(s = \text{personal}(u)))$ {
      Remove the edge from s to k in $PS.G$.
    } **else** {
      $c \leftarrow c \cup \{s\}$.
    }
  }
  if $(|c| = 0)$ { ▹ Discard k and unused ancestors
    Let s be an empty stack.
    Push k onto s.
    while (s is not empty) {
      Pop r off s.
      if (there are no incoming edges to r in $PS.G$) {
        for each (r' such that there is an edge from r to r' in $PS.G$) {

[15] It is not possible for a member to unilaterally "leave" the group, because there is no way for it to provably generate a new group key without learning enough information to derive the key. Section 10.12 revisits this subject and describes an approach that achieves a similar effect.

```
                Push r′ onto s.
            }
            Execute KGM.DiscardKey with:
              ▸ Parameter k: set to r.
        }
    }
} else if (|c| = 1 && KC.MayMerge(PS.G, s, k) = 1 for s ∈ c) {          ▹ Merge s into k
    Let s be the single element of c.
    Set pk such that the mapping s → pk is in PS.P.
    Update the mapping k → pk in PS.P.
    if (SS ≠ ⊥) {
        if (there is a mapping from s in SS.Q) {
            Set sk such that the mapping s → sk is in SS.Q.
            Update the mapping k → sk in SS.Q.
        } else {
            Delete any mapping from k in SS.Q.
        }
    }
    Delete any mapping to k from PS.S.
    for each (u → s in PS.S) {
        Add the mapping u → k to PS.S.
    }
    if (s ∈ PS.X) { Add k to PS.X. }
    else if (k ∈ PS.X) { Remove k from PS.X. }
    Execute KGM.DiscardKey with:
      ▸ Parameter k: set to s.
} else {                                                  ▹ Transfer a new key pair for k
    Execute KGM.NewKeyTransfer with:
      ▸ Parameter R: set to c
      ▸ Parameter bctx: set to bctx
      ▸ Output pk*: store in pk*
      ▸ Output sk*: store in sk*
      ▸ Output bctx: store in bctx.
    Update the mapping k → pk* in PS.P.
    if (k ∈ PS.X) { Remove k from PS.X. }
    if (SS ≠ ⊥) {
        if (sk* = ⊥) {
```

```
                Delete any mapping from k in SS.Q.
            } else {
                Update the mapping k → sk* in SS.Q.
            }
        }
        Delete any mapping to k from PS.S.
        for each (r ∈ c) {
            for each (u such that u → r is in PS.S) {
                if (k ∉ keyset_{PS.G}(u)) { Add the mapping u → k to PS.S. }
            }
        }
        if (U_perf ∉ userset_{PS.G}(()k)) {
            Add the mapping U_perf → k to PS.S.
        }
    }
}
return bctx.
```

10.5.8 Refresh Keys

The **KGM.RefreshKeys** operation updates all of the exhausted keys so that they are no longer exhausted. The group key is also updated using one of two unique KDFs—denoted KDF_0 and KDF_1—based on whether or not the old group key is used in the derivation.

Parameters: R, a set of k-nodes such that every member knows at least one corresponding private key; b, a bit indicating whether or not the old group key should be required to derive the new group key; and *bctx*, a CBRAKEM update context.
Output: *bctx*, the updated CBRAKEM context.

---

if (there exists a member U such that $\text{keyset}_{PS.G}(U) \cap R = \emptyset$) { Abort. }
Execute KGM.NewKeyTransfer with:

- Parameter R: set to R
- Parameter *bctx*: set to *bctx*
- Output pk^*: store in pk^*
- Output sk^*: store in sk^*
- Output *bctx*: store in *bctx*.

```
for each (k ∈ PS.X) {
    Set pk such that the mapping k → pk is in PS.P.
    pk′ ← MixPK(pk, pk*).
    Update the mapping k → pk′ in PS.P.
    if (SS ≠ ⊥ && there is a mapping from k in SS.Q) {
        Set sk such that the mapping k → sk is in SS.Q.
        sk′ ← MixSK(sk, sk*).
        Update the mapping k → sk′ in SS.Q.
    }
}
PS.X ← ∅.
if (SS ≠ ⊥) {
    if (b = 0) {
        SS.K ← KDF0(sk*||PS.H).
    } else {
        SS.K ← KDF1(SS.K||sk*||PS.H).
    }
}
return bctx.
```

10.5.9 Ratchet Group Key

The **KGM.RatchetGroupKey** operation simply updates the group key using a unique KDF. This operation can be used to provide a very efficient form of forward secrecy.

```
if (SS ≠ ⊥) {
    SS.K ← KDF(SS.K||PS.H).
}
```

10.5.10 Share Key

The **KGM.ShareKey** operation uses CBRAKEM to transfer the private key for a k-node to a given set of recipient k-nodes. The key graph is updated to reflect the knowledge transfer.

Parameters: k, a k-node to share; R, a set of k-nodes to receive the private key; and *bctx*, a CBRAKEM update context.
Output: *bctx*, the updated CBRAKEM context.

---

if (k is a personal key) { Abort. }
Set pk^* such that the mapping $k \rightarrow pk^*$ is in $PS.P$.
if (performer context) {
    if (there is no mapping from k in $SS.Q$) { Abort. }
    Set sk^* such that the mapping $k \rightarrow sk^*$ is in $SS.Q$.
} **else** {
    $sk^* \leftarrow \perp$.
}
Execute KGM.KeyTransfer with:
- Parameter R: set to R
- Parameter pk^*: set to pk^*
- Parameter sk^*: set to sk^*
- Parameter *bctx*: set to *bctx*
- Output sk^*: store in sk^*
- Output *bctx*: store in *bctx*.

if ($sk^* \neq \perp$) { Store mapping $k \rightarrow sk^*$ in $SS.Q$. }
for each ($s \in R$) {
    if (there is no path from s to k in $PS.G$) {
        Add an edge from s to k in $PS.G$.
    }
}
return *bctx*.

As discussed in Section 10.5.6, checking for the existence of a path from s to k can be implemented efficiently with a depth-first search; the Safehouse configurations used in practice do not require a more efficient solution.

10.5.11 Import Tree

The **KGM.ImportTree** operation imports the key tree produced by a TKLL protocol execution (see Section 9.4) into the key graph. None of the new nodes are connected to any of the existing nodes. The processing agent may optionally provide private keys for nodes in the key tree if it

was one of the participants in the TKLL protocol; these will be imported into the private state. In general, the set of private keys given as a parameter will be different for each processing agent.

Parameters: T, the key tree instantiated by the TKLL protocol execution; P^*, a bijection from k-nodes in T to public keys; and Q^*, a mapping from k-nodes in T to private keys.

---

if (any u-node in T has the same agent ID as a u-node in $PS.G$) { Abort. }
Copy all of the k-nodes, u-nodes, and edges from T into $PS.G$.
Copy all of the mappings from P^* into $PS.P$.
if (Q^* is not empty) {
    if ($SS = \perp$) {
        $SS.K \leftarrow \perp$.
        $SS.Q \leftarrow \emptyset$.
        $SS.E \leftarrow \perp$.
        $SS.U \leftarrow \perp$.
    }
    Copy all of the mappings from Q^* into $SS.Q$.
}

In real implementations, k-nodes may be identified by distinct numeric labels. It is the responsibility of KGM.ImportTree to appropriately relabel k-nodes imported from T (while preserving the structure) in order to ensure that there are no collisions in the resulting identifiers.

10.6 Key Control Overview

When a Safehouse session is created, a KC implementation must be selected. As discussed in Section 10.1.3, the KC subsystem is responsible for determining the shape of the key graph, $PS.G$.

A KC scheme provides two functions that are used to initialize a new Safehouse session:

- **KC.InitGraphPerform**$(U) \rightarrow (G, P, Q, T)$: a function that initializes the key graph for a new Safehouse session created by an initial member with agent ID U. The function produces the initial key graph G with associated public key map P and private key map Q. It also produces an *initialization message* T, which can be consumed by the server to derive the same key graph.
- **KC.InitGraphApply**$(U, T) \rightarrow (G, P)$: a function that consumes an initialization message T produced by an initial member U and produces the resulting key graph G and public key

map P. This function is only executed by the server during the arranged creation of a new Safehouse session.

The KC.InitGraphPerform and KC.InitGraphApply operations are provided by the KC subsystem because the shape of the initial key graph determines the most efficient way to construct the initialization message. These functions are used as part of an MLS-layer operation that initializes the complete group state, discussed in Section 10.10.2.

The KC scheme is also responsible for keeping track of shape-specific structural data as the key graph evolves. For example, if the KC scheme shapes the graph into a tree structure, additional data may be stored that tracks tree statistics in order to inform decisions about where to add new k-nodes. As the KGM-layer operations described in Section 10.5 modify the key graph, the KC scheme updates its data accordingly. This process is not shown in the KGM operation or KC scheme definitions because it is mainly an implementation concern, since the KC implementations discussed in this chapter can always derive the data that they need from a generic graph structure.

The KC scheme provides a function that determines whether or not it is acceptable to merge k-nodes during a KGM.Evict operation (see Section 10.5.7):

- **KC.MayMerge**$(G, s, k) \rightarrow b$: a function that takes two k-nodes, s and k, in the key graph G as input, and determines whether or not the k-nodes may be merged. It must be the case that an edge exists from s to k in G. The output is a bit b, where $b = 1$ indicates that a merger is permitted.

In practice, KC.MayMerge may be implemented as a stateful function that references the structural data about G previously extracted and tracked by the KC scheme.

There are several locations in Safehouse where it is necessary to serialize or deserialize the public state. This is mainly done when a new member joins the session, because they must first download the existing public state before they can add themselves using a joining operation. The public state is also uniquely serialized when it is used as the input to a hash function.[16]

[16] It is interesting to note that in general, outside of the context of Safehouse, using an abstract object as the input to a cryptographic hash function requires only a serialization procedure. The serialization must be unique in order for the hash output to be deterministic. However, if a matching lossless deserialization function does not *exist*, then this is a *security flaw* instead of a *correctness bug*. This is because if the serialization procedure is lossy, then "collisions" can be found by altering only the information about the object that is lost during serialization; the input to the hash function, and therefore its output, will be the same. A classic example of this flaw occurred in the PGP 2.x software when the sizes of the RSA modulus and encryption exponent were lost during serialization [Ley02]. The existence of a lossless deserialization function ensures that collisions cannot be produced in this way. Since Safehouse requires a lossless deserialization function for actual transmission purposes, the serialization function is also safe to use when producing hash inputs.

Serialization and deserialization of large parts of the public state are the responsibility of the KC subsystem because the size of the encoded data can be greatly reduced by taking advantage of the key graph structure. A generalized implementation for serializing the key graph would need to transmit a uniquely ordered list of u-nodes, k-nodes, edges, and the corresponding contents of the other elements of PS. In contrast, when both the serializer and deserializer know about invariants that apply to the shape of the graph (e.g., knowing that the graph is a tree or that it is a forest with a maximum number of leaves in each tree), some node or edge data can be omitted from the serialization. The KC scheme provides the following two functions to implement serialization and deserialization:

- **KC.Serialize**$(G, P, S, I) \rightarrow D$: a function that accepts a key graph G with the expected shape, a mapping from k-nodes to public keys P, a bipartite graph of superfluous edges S, and an identity table I, and losslessly produces a unique opaque binary blob D.
- **KC.Deserialize**$(D) \rightarrow (G, P, S, I)$: a function that accepts a binary blob D and losslessly decodes it to produce G, P, S, and I as defined in KC.Serialize.

The KC scheme is responsible for serializing the contents of $PS.I$ because this is the only portion of the public state that binds data to members (see Section 10.1.2). Since the KC scheme has knowledge about the graph structure, it can strategically position the blinded identity tuples in the serialization based on the location of the corresponding u-nodes in the key graph. Like KC.MayMerge, in practice KC.Serialize may statefully reference structural data rather than dynamically reconstructing it from G. However, KC.Deserialize must not require state, because it must be usable by agents that have no prior knowledge of the group state. Note that the serialization functions assume that $PS.X = \emptyset$; this is acceptable because serialization only occurs for safe states with no exhausted keys.

The KC scheme provides two functions that return certain k-nodes in the graph based on structural knowledge:

- **KC.BroadcastKeys**$(G) \rightarrow K$ returns a set of k-nodes K such that every member has access to at least one corresponding private key. Formally, $\cup_{k \in K} \text{userset}_G(k) = \text{userset}(G)$.
- **KC.SharedKey**$(G) \rightarrow k$ returns a k-node such that $\text{userset}_G(k) = \text{userset}(G)$, or $\perp$ if none exists.

These two functions exist in order to improve the practicality of the system; general algorithms exist to extract this information from arbitrary key graphs, but they involve solving small NP-

complete problem instances.[17] In practice, these functions can use stateful structural information to locate the requested k-nodes in constant time, which avoids the need to use the potentially expensive general algorithms.

Finally, the KC scheme provides two components that are necessary to implement Safehouse's mass join functionality. The first component is an interactive protocol called **KC.PrepareJoin** that is executed by one or more new members, producing a *joining commit*. The second is an operation called **KC.MergeJoin** that takes a joining commit as a parameter and adds the members to the session.

The **KC.PrepareJoin** protocol accepts the following parameters from the caller:

- The current public state for the Safehouse session, PS.
- The caller's agent ID, which must not yet be a group member.
- A set of agent IDs for the *joining members*, including the caller. The agent IDs must be distinct and must not already be present in $PS.G$.
- A mapping from the joining members to public keys produced by CBRAKEM.KeyGen, which will be bound to their personal keys in the key graph.
- A mapping from the joining members to the invitation in $PS.L$ that they are using to join the session.
- The caller's ephemeral private key produced by CBRAKEM.KeyGen, which will be bound to its personal key in the key graph.
- The caller's (unblinded) identity public key.
- The caller's identity witness.

[17] A general approach for implementing KC.BroadcastKeys, assuming that the list of u-nodes is easily accessible, is to choose an arbitrary u-node and to recursively follow outgoing edges until a set of nodes with an outdegree of zero is found. For each of these "terminal nodes", the edges can be explored in reverse to identify the u-nodes with paths to the node (i.e., the user set). A minimal subset of the terminal nodes is selected such that the union of the user sets is equal to the union of all terminal nodes' user sets; this is the NP-complete *set cover problem*. This subset of terminal nodes is added to the broadcast key set, and the u-nodes in the corresponding user sets are marked as complete. The process is repeated until all u-nodes are marked as complete, starting by selecting another arbitrary u-node that is not marked as complete. KC.SharedKey can be implemented by simply checking to see if the output of KC.BroadcastKeys has a single element and returning that element, or returning $\perp$ otherwise.

As the protocol is executed, multiple transmissions of opaque data blobs may be sent to subsets of the joining members; the developer must provide a mechanism to deliver these transmissions. When the protocol completes successfully, it outputs the joining commit and *joining private data*; both are opaque data blobs from the perspective of other subsystems. The same joining commit is output for all joining members, whereas the joining private data is unique for each joining member. In practice, most KC schemes implement KC.PrepareJoin using the authenticated TKLL protocol (see Section 9.4).

The joining members produce authenticating NIZKPKs showing that they are permitted to join the group as part of the KC.PrepareJoin protocol. These NIZKPKs are included in the joining commit and verified by the server and existing members when applying the commit; see Section 10.8 for details about the authentication mechanism.

The **KC.MergeJoin** operation is performed as part of an MLS-layer operation by one of the joining members, with the result that all joining members are added to the session. The operation's parameters are a joining commit produced by KC.PrepareJoin and the joining private data. Joining members provide the joining private data produced by their own call to KC.PrepareJoin, whereas other agents set this parameter to $\perp$. Initially, the joining members' intermediate private state is $\perp$ when KC.MergeJoin is executed. Afterwards, the joining members are left with complete private states that transform them into members of the session.

Several KC schemes are described in Section 10.11.

10.7 Label-Value Store Manipulation

The Safehouse group state contains three different label-value stores: the public label-value store *PS.A*, the confidential label-value store *PS.E*, and the unblinding ciphertext table *PS.U*. A high-level description of the label-value stores and how they operate was given in Section 10.1.7 for *PS.A* and *PS.E*, and in Sections 10.1.8 and 10.1.9 for *PS.U*. The public label-value store *PS.A* is trivial: it is simply a list of unencrypted label-value pairs. The state hash, discussed in Section 10.1.3, enables Safehouse to ensure the integrity of *PS.A*, which is all that is required. Since there is no cryptography involved with accessing or altering *PS.A*, it is not discussed further in this section. In contrast, *PS.E* and *PS.U* require a variety of cryptographic mechanisms to implement. Whereas *PS.E* provides a developer-facing confidential storage mechanism for arbitrary values, *PS.U* is only used internally by Safehouse to implement anonymity preservation by storing private keys used to unblind identity public keys. Nonetheless, *PS.E* and *PS.U* share large parts of their implementation. This section discusses the motivating factors behind the general design, and Section 10.7.1 describes a specific implementation.

As discussed in Section 10.1.7, the contents of the confidential stores must be decryptable in two scenarios: new members must be able to non-interactively decrypt the contents upon joining the session, and existing members must be able to decrypt new values when they are updated. Since the only private key material available to new members is an invitation private key (see Section 10.1.9), and no interactivity with existing members can be used to establish new private key material, the keys necessary to decrypt the stores must necessarily be encrypted to all invitation public keys associated with entries in the current layaway table, *PS.L*. Moreover, since existing members likely do not have access to a currently valid invitation private key (especially since individual invitations are removed from the layaway table upon use), value updates must also be encrypted to all existing members. Encrypting the keys is accomplished by using the chosen BRAKEM scheme, discussed in Chapter 8, to produce a single encapsulation to a single receiver set. This BRAKEM encapsulation can then be stored in the public state.

Forward secrecy is a concern for the values stored in *PS.E* and *PS.U*, but the attack scenarios are slightly different than they are for the group key. It is important to consider these scenarios in order to avoid an over-engineered design that pays a performance price for no real security gains. The only legitimate mechanism for decrypting values *currently* stored in *PS.E* or *PS.U* is to use an invitation private key to decrypt the ciphertexts. An honest server does not store historical public states for a session. This means that if the server is honest and the adversary compromises an invitation private key, they will only be able to decrypt the currently stored values, which is as expected. However, an honest-but-curious server is free to store historical public states. In this scenario, if the adversary acquires a historical public state and an invitation private key associated with an entry in the state's layaway table, then they will be able to recover a historical value; there is no way to avoid this attack without either requiring an interactive protocol with an existing member upon joining, or not giving new members immediate access to the values. Consequently but perhaps unintuitively, there is usually no benefit to re-encrypting values under a new key unless the value changes, since the easiest way for an adversary to recover the value is usually to attack an old public state in cooperation with the server.

Finally, efficiency is an important consideration in the design of the label-value stores. The simplest solution based on the aforementioned design considerations would be to use BRAKEM to encrypt each value to each invitation public key in the layaway table and to the existing members. However, this design incurs a large performance cost when issuing an invitation (which would require all values to be encrypted to new invitation public key) or when adding a value to *PS.E* or *PS.U* (which would require the value to be encrypted to all invitation public keys). These costs can be mitigated by introducing an intermediate private key to "decouple" changes to either list. This is the purpose of the *anchor key pair* consisting of the *anchor public key*, *apk*, and the *anchor private key*, *ask*, as first mentioned in Section 10.1.9. The anchor private key is encrypted to all of the invitation public keys using BRAKEM, and all entries in the store are encrypted to the anchor

public key using BRAKEM. When the layaway table is updated, the anchor private key can simply be encrypted to the new list of invitation public keys, without altering the ciphertexts for the values; this is called a **RefreshInvites** operation. If the anchor key pair is changed, then all of the values only need to be re-encrypted to the new anchor public key, rather than to all of the invitation public keys; this is called a **RefreshEncryption** operation.

A separate anchor key pair is used for *PS.E* and *PS.U*, since it is usually not necessary to use the RefreshEncryption operation for both stores at the same time. In general, using separate anchor key pairs means that fewer RefreshEncryption operations are required. The anchor private key for *PS.E* is stored in *SS.E.A*, and the anchor private key for *PS.U* is stored in *SS.U.A*. Members continue to store the current anchor private keys even after joining the session.

A malicious insider can always provably share the decrypted values with the adversary. Safehouse tracks when an anchor key pair is *fresh*, meaning that the adversary can only recover the anchor private key with the help of a malicious insider. The anchor key pair becomes *stale* when the adversary could recover the anchor private key by compromising a former member or an invitation private key that is no longer valid; this occurs when a member is evicted from the session, keys are updated for post-compromise security purposes, or an invitation is retracted. When the anchor key pair is stale, a RefreshEncryption operation must be performed before new values are stored. This prevents the adversary from learning any more than it could by compromising an old group state, as discussed above.

The LVSM layer provides all of the low-level operations necessary to access and update *PS.E* and *PS.U*. In summary, the layer provides the following operations for both stores:

- **Initialize** sets up an empty label-value store. It is used only as part of the MLS-layer operation that initializes the group state at the start of a session.
- **RefreshInvites** encrypts the current anchor private key to all of the invitation public keys in the layaway table, storing the BRAKEM encapsulation in the public state.
- **RefreshEncryption** generates a new anchor key pair, encrypts the anchor private key to all of the invitation public keys in the layaway table as well as all existing members, re-encrypts all of the values to the new anchor public key, and stores all of the resulting BRAKEM encapsulations and the anchor public key in the public state. NIZKPKs are included as needed so that the correctness of the operation is publicly verifiable.
- **MarkStale** marks the anchor key pair as stale.
- **Set** changes a value associated with a given label by encrypting the new value to the anchor public key and storing the BRAKEM encapsulation in the public state.

- **Delete** removes a value associated with a given label by deleting the ciphertext and label in the public state.

These operations are provided for *PS*.*E* with the prefix **LVSM**.*E*, and for *PS*.*U* with the prefix **LVSM**.*U*. For example, LVSM.*E*.Initialize denotes the operation that initializes *PS*.*E*. In addition to these operations, a decryption function is provided for both *PS*.*E* and *PS*.*U* with the same prefixes:

- **Decrypt**$(PS, SS, id, isk) \rightarrow SS'$ decrypts the contents of the store in the given public state, updating the given private state, using a given invitation private key *isk* with invitation identifier *id*. The function returns $\perp$ if the decryption failed.

The LVSM.*E*.Decrypt and LVSM.*U*.Decrypt functions are used by new members when joining the session to gain access to the confidential data in the stores. Existing members do not use these functions; when an anchor key pair changes as part of a RefreshEncryption operation, or when the content of the stores is changed by a Set or Delete operation, existing members' private states are updated as part of the algorithm to apply the operation.

BRAKEM is used to encrypt the keys because it is necessary for the LVSM operations to be publicly verifiable. However, this is not enough to guarantee that the values themselves are correctly formed. In the *PS*.*U* case, additional NIZKPKs are used when authenticating identities (as discussed in Section 10.8) in order to show that the encrypted values in *PS*.*U* will correctly unblind the identity public key. In general, this cannot be done for the developer-supplied values stored in *PS*.*E*. Safehouse is not aware of the required structure for the values stored in *PS*.*E*, so it cannot provide any guarantees about the correctness of the values. The design takes advantage of this limitation by using efficient but unverifiable cryptosystems to protect the values. For this reason, secure group messaging tools built using Safehouse must be able to handle values in *PS*.*E* that have been covertly corrupted by malicious insiders.

The implementation of the LVSM layer depends on the developer's choice of BRAKEM construction for a Safehouse deployment. In any case, the LVSM layer is implemented as described above. Section 10.7.1 provides specific implementation details in the case that $\text{BRAKEM}_{\star}^{\text{DDL}}$ is used. The same general approach can be used to implement the LVSM layer for $\text{BRAKEM}^{\text{ZK}}$.

10.7.1 Implementing with Double Discrete Logarithms

This section describes how to implement the LVSM layer introduced in Section 10.7 when using $\text{BRAKEM}_{\star}^{\text{DDL}}$, as discussed in Section 10.2. Section 10.7.1.1 describes how to implement the

operations for the confidential label-value store, Section 10.7.1.2 introduces a new NIZKPK used to implement the confidential label-value store, and Section 10.7.1.3 describes how to implement the operations for the unblinding ciphertext table. Whenever operations in this section read elements from the commit, they are implicitly assumed to validate that the elements belong to the expected group (if any), aborting if this is not the case.

10.7.1.1 Confidential Label-Value Store

The confidential label-value store is implemented in mostly the manner described in Section 10.7, but with three important optimizations:

- Instead of directly using the anchor key pair, which is necessarily generated from the large and slow $\text{BRAKEM}_{\star}^{\text{DDL}}$ key space, the anchor private key *ask* is also used to produce a public key in an elliptic curve group, *cpk*. The private key is proven to be equivalent using the standard Schnorr-based DLEQ NIZKPK presented in Section 8.2.2.2. It is possible to use this efficient NIZKPK securely as long as elliptic curve group has order p_3.[18]
- Since the values do not need to be verifiably correct, it is possible to encrypt the values using a standard symmetric cryptosystem. For each value, a secret random curve point called the *entry key* is encrypted to *cpk*. Each value is encrypted using authenticated encryption with the symmetric key derived from the associated entry key using a unique KDF. $\text{AEnc}_k(v)$ denotes the encryption of v with key $\text{KDF}(k)$ using the authenticated encryption scheme. $\text{ADec}_k(c)$ denotes the corresponding decryption operation for ciphertext c with key $\text{KDF}(k)$, which may return $\perp$ if the decryption fails.
- The mechanism for encrypting the entry keys needs to be a symmetric cryptosystem using *ask* as the key, but there must also exist an efficient plaintext equality NIZKPK for the cryptosystem. The existence of this efficient NIZKPK allows the RefreshEncryption operation to efficiently re-encrypt the same entry keys to a new anchor key pair without requiring the values themselves to be re-encrypted. This allows very large ciphertexts to be stored in $PS.E$ with no size-based performance penalties, except when updating the large value. Entry key encryption is accomplished using elliptic curve ElGamal in a pseudo-symmetric mode: the private key *ask* is used at both encryption and decryption times. The "ElGamal randomness" public key is derived from a random hash stored alongside the value.

Figure 10.2 depicts the structure of $PS.E$ when using $\text{BRAKEM}_{\star}^{\text{DDL}}$ with the aforementioned optimizations.

[18] Otherwise, the proof would be unsound. This soundness problem was discussed in Section 8.2.3.

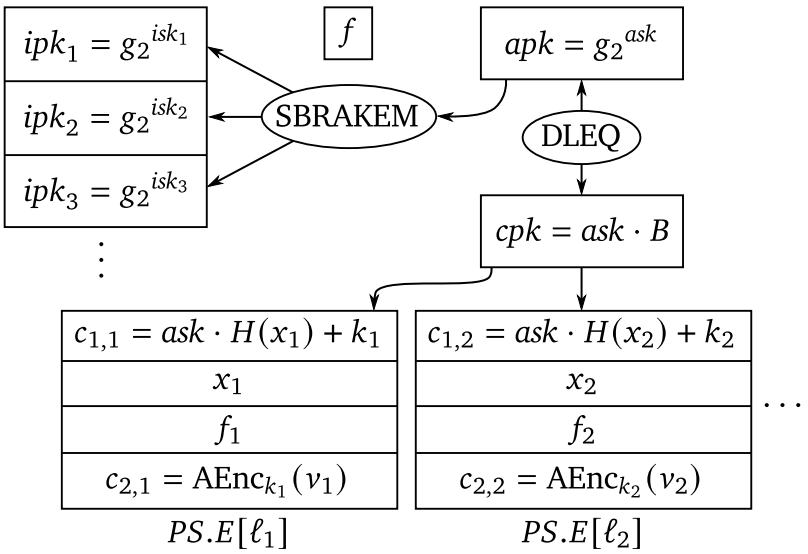


Figure 10.2 THE CONFIDENTIAL LABEL-VALUE STORE STRUCTURE WHEN USING $\text{BRAKEM}_\star^{\text{DDL}}$. The anchor private key *ask* is transferred to the invitation public keys in *PS.L* using $\text{BRAKEM}_\star^{\text{DDL}}$. The anchor public key, *apk*, is proven to be equivalent to an elliptic curve version, *cpk*, using a DLEQ NIZKPK. Elliptic curve ElGamal is used to encrypt the entry key k_i for the i^{th} label-value pair, which enables efficient Schnorr plaintext equality NIZKPKs when re-encrypting k_i. A standard authenticated encryption scheme is used to encrypt the value for an entry using the associated entry key. f indicates the anchor key pair's freshness, while f_i indicates the freshness of entry key i. (Ref: 368)

The prototype Safehouse implementation uses the sample parameters presented in Section 8.3.1.4 for its BRAKEM$^{\text{DDL}}_{\star}$ implementation. These parameters set p_3 to the order of the Curve25519 [Ber06] elliptic curve group. However, the LVSM implementation does not use Curve25519 directly. Instead, it uses Ristretto [VGT+21] as the elliptic curve group; this group has the same order as Curve25519, but comes with several practical safety advantages. B denotes the Ristretto base point. H denotes a secure "hash to curve" function; for Ristretto, this is usually a cryptographic hash with the output processed by the Elligator 2 method [FSS+20, §6.8.2].

The entry key for entry i is denoted as k_i.[19] Each entry key k_i is an element in the Ristretto group that is generated by using H to hash a 128-bit value chosen uniformly at random.[20] Although k_i can be efficiently and verifiably re-encrypted to a new *cpk*, k_i itself will remain stale (i.e., accessible to an adversary that compromises outdated private key material). Consequently, the freshness of the entry keys must be trackable independently of the freshness of the anchor key pair. $PS.E.f$ stores a bit that indicates if *ask* is fresh; it is fresh if $f = 1$. Similarly, k_i is fresh if the entry-specific freshness bit $f_i = 1$.

In summary, the scheme stores the following data in the public state:

- $PS.E.T$: an SBRAKEM encapsulation of $SS.E.ask$ to the set of invitation public keys in $PS.L$.
- $PS.E.apk$: the anchor public key in $\mathbb{G}_2$.
- $PS.E.cpk$: the anchor public key in the Ristretto group.
- $PS.E.\pi$: a DLEQ proof showing that *apk* and *cpk* have the same discrete logarithm in their respective groups.
- $PS.E.f$: a bit indicating if *ask* is fresh.
- For the i^{th} entry with label ℓ_i:
  - $PS.E[\ell_i]$: a tuple $(f_i, x_i, c_{1,i}, c_{2,i})$.
  - f_i: a bit indicating if the entry key k_i is fresh.
  - x_i: a random 128-bit value.
  - $c_{1,i}$: an ElGamal encryption of k_i to *cpk* using ElGamal random public key $H(x_i)$.

[19] Since the values are indexed by the associated label, the entries are not normally addressable by position. However, this section uses integer indices as a notational convenience for explanatory purposes.

[20] This procedure generates a group element with an unknown discrete logarithm (with respect to the base B) uniformly at random. The randomly generated 128-bit value that is used as input to H is an intermediate value that is not stored.

- $c_{2,i}$: an authenticated encryption of the value v_i using key $\text{KDF}(k_i)$.

The following data is stored in the private state:

- $SS.E.ask$: the anchor private key.
- For the i^{th} entry with label ℓ_i:
  - $SS.E[\ell_i]$: the decrypted value v_i.

In practice, implementations may cache intermediate keys, such as the entry keys. The following sections describe how to implement the LVSM operations and the decryption function for $PS.E$.

10.7.1.1.1 Initialize

The **LVSM.E.Initialize** operation proceeds as follows:

```
Execute CKeyGen with:                                          ▹ Generate anchor key pair
  ▸ Output pk*: store in PS.E.apk
  ▸ Output sk*: store in SS.E.ask.
if (performer context) {                                       ▹ Generate anchor on curve
    PS.E.cpk ← ask · B.
    Write PS.E.cpk to the commit.
} else {
    Read PS.E.cpk from the commit.
}
if (performer context) {                                       ▹ Prove anchors are equivalent
    Prove π: PK{(ask) : apk = g_2^ask ∧ cpk = ask · B}.
    Write π to the commit.
} else {
    Read π from the commit.
    if (π fails to verify) { Abort. }
}
PS.E.π ← π.
PS.E.f ← 1.
```

10.7.1.1.2 Refresh Invitations

The **LVSM.*E*.RefreshInvites** operation proceeds as follows:

Let R denote the set of invitation public keys stored in $PS.L$.
if (performer context) {
    $T \leftarrow$ SBRAKEM.Encapsulate$(R, PS.E.apk, SS.E.ask)$.
    Write T to the commit.
} **else** {
    Read T from the commit.
    if (SBRAKEM.Verify$(R, PS.E.apk, T) \neq 1$) { Abort. }
}
$PS.E.T \leftarrow T$.

10.7.1.1.3 Refresh Encryption

The **LVSM.*E*.RefreshEncryption** operation proceeds as follows:

$apk' \leftarrow PS.E.apk$.
if $(SS \neq \perp)$ { $ask' \leftarrow SS.E.ask$. }
Execute LVSM.E.Initialize. ▹ Update anchor key pair
Execute LVSM.E.RefreshInvites. ▹ Transfer anchor private key to invitations
Let R' = KC.BroadcastKeys$(PS.G)$. ▹ Transfer anchor private key to members
if (performer context) {
    $T' \leftarrow$ SBRAKEM.Encapsulate$(R', PS.E.apk, SS.E.ask)$.
    Write T' to the commit.
} **else if** $(SS = \perp)$ {
    if (SBRAKEM.Verify$(R', PS.E.apk, T') \neq 1$) { Abort. }
} **else** {
    Find $r \in R'$ such that $r \rightarrow sk$ is in $SS.Q$.
    Let j represent the position of r in R'.
    $SS.E.ask \leftarrow$ SBRAKEM.Decapsulate$(R', j, sk, PS.E.apk, T')$.
    if $(SS.E.ask = \perp)$ { Abort. }
}
for each (entry $(f_i, x_i, c_{1,i}, c_{2,i})$ with label ℓ_i in $PS.E$, sorted) { ▹ Re-encrypt entry keys
    $x'_i \leftarrow x_i$. ▹ Store old values for use in the NIZKPK
    $c'_{1,i} \leftarrow c_{1,i}$.

```
    if (performer context) {
        k_i ← c_{1,i} − ask′ · H(x_i).
        x_i ←$ [0, 2^128).
        c_{1,i} ← ask · H(x_i) + k_i.
        Write x_i and c_{1,i} to the commit.
    } else {
        Read x_i and c_{1,i} from the commit.
    }
    PS.E[ℓ_i] ← (f_i, x_i, c_{1,i}, c_{2,i}).
}
if (performer context) {                                   ▷ Prove correctness
    Produce a NIZKPK π showing that all updates are correct.
    Write π to the commit.
} else {
    Read π from the commit.
    if (π fails to verify) { Abort. }
}
PS.E.f ← 1.
```

The NIZKPK produced as part of the operation, π, must prove the following:

- knowledge of ask', the discrete log of the old anchor public key, apk';
- knowledge of ask, the discrete log of the new anchor public key, apk; and
- that for every label-value pair (denoted with index i), the old entry key was encrypted to the new anchor public key.

If there are m entries in the store with indices 1 to m, then the proof corresponds to the following statement, given in Camenisch-Stadler notation [CS97]:

$$\begin{aligned} PK\{(ask, ask') : \; & apk' = ask' \cdot B \; \wedge \\ & apk = ask \cdot B \; \wedge \\ & \forall_{i=1}^{m} \; c'_{1,i} - c_{1,i} = ask' \cdot H(x'_i) - ask \cdot H(x_i) \; \} \end{aligned}$$

This proof can be implemented using techniques similar to those used by the BDLEQ NIZKPK presented in Section 8.2.2.3. A NIZKPK for the statement is introduced in Section 10.7.1.2, along with security proofs.

10.7.1.1.4 Mark Stale

The **LVSM.E.MarkStale** operation proceeds as follows:

$PS.E.f \leftarrow 0$.
for each (entry $(f_i, x_i, c_{1,i}, c_{2,i})$ with label ℓ_i in $PS.E$) {
    $PS.E[\ell_i] \leftarrow (0, x_i, c_{1,i}, c_{2,i})$.
}

10.7.1.1.5 Set

The **LVSM.E.Set** operation proceeds as follows:

Parameters: ℓ, a label; and v, either a value to store or $\perp$.

---

if ($PS.E.f = 0$) {
    Execute LVSM.E.RefreshEncryption.
}
if ($PS.E[\ell]$ exists) {
    $(f_i, x_i, c_{1,i}, c_{2,i}) \leftarrow PS.E[\ell]$.
} **else** {
    $f_i \leftarrow 0$.
}
if ($f_i \neq 1$) { ▹ If stale, generate a new entry key
    if (performer context) {
        Set k_i to a curve point uniformly at random.
        $x_i \xleftarrow{\$} [0, 2^{128})$.
        $c_{1,i} \leftarrow SS.E.ask \cdot H(x_i) + k_i$.
        Write x_i and $c_{1,i}$ to the commit.
    } **else** {
        Read x_i and $c_{1,i}$ from the commit.
    }
} **else if** (performer context) {
    $k_i \leftarrow c_{1,i} - SS.E.ask \cdot H(x_i)$.
}
if (performer context) { ▹ Encrypt the new value
    if ($v = \perp$) { Abort. }

$c_{2,i} \leftarrow \text{AEnc}_{k_i}(v)$.
Write $c_{2,i}$ to the commit.
} **else** {
if ($v \neq \perp$) { Abort. }
Read $c_{2,i}$ from the commit.
if ($SS \neq \perp$) {
$k_i \leftarrow c_{1,i} - SS.E.ask \cdot H(x_i)$.
$v \leftarrow \text{ADec}_{k_i}(c_{2,i})$.
$SS.E[\ell] \leftarrow v$.
}
}
$f_i \leftarrow 1$. ▷ Update the store
$PS.E[\ell] \leftarrow (f_i, x_i, c_{1,i}, c_{2,i})$.

The operation is performed with parameter v set to the new value, and received with $v = \perp$. Members can retrieve the value from $SS.E[\ell]$ after receiving the operation. It is important to note that the performer does not produce a NIZKPK showing that v was correctly formed. For this reason, it is also not necessary to prove that $c_{2,i}$ is valid. In such situations, $SS.E[\ell]$ will end up set to $\perp$. Since the server always has $SS = \perp$, it can never detect this condition, and so it is not possible for members to reject the operation when they detect a malformed $c_{2,i}$—doing so would fork the session. Consequently, the developer must be able to handle the scenario in which $SS.E[\ell] = \perp$.

10.7.1.1.6 Delete

The **LVSM.E.Delete** operation proceeds as follows:

Parameters: ℓ, a label.

---

if ($PS.E[\ell]$ exists) { Delete $PS.E[\ell]$. }
if ($SS \neq \perp$) {
if ($SS.E[\ell]$ exists) { Delete $SS.E[\ell]$. }
}

10.7.1.1.7 Decrypt

The **LVSM.E.Decrypt**$(PS, SS, id, isk) \rightarrow SS'$ function proceeds as follows:

Let R denote the set of invitation public keys stored in $PS.L$.
Let j refer to the invitation in R with identifier id.
$SS' \leftarrow SS$.
$SS'.E.ask \leftarrow \text{SBRAKEM.Decapsulate}(R, j, isk, PS.E.apk, PS.E.T)$.
if $(SS'.E.ask = \perp)$ { **return** $\perp$. }
if ($PS.E.\pi$ fails to verify) { **return** $\perp$. }
for each (entry $(f_i, x_i, c_{1,i}, c_{2,i})$ with label ℓ_i in $PS.E$) {
    $k_i \leftarrow c_{1,i} - SS'.E.ask \cdot H(x_i)$.
    $v \leftarrow \text{ADec}_{k_i}(c_{2,i})$.
    $SS'.E[\ell] \leftarrow v$.
}
return SS'.

10.7.1.2 NIZKPK for Refreshing Encryption

The LVSM.E.RefreshEncryption operation introduced in Section 10.7.1.1.3 requires a NIZKPK to prove that the same entry keys were correctly re-encrypted to the new anchor public key with elliptic curve ElGamal. For convenience, the proof statement is repeated here:

$$\begin{aligned} PK\{(ask, ask') : \; & apk' = ask' \cdot B \; \wedge \\ & apk = ask \cdot B \; \wedge \\ & \forall_{i=1}^{m} \; c'_{1,i} - c_{1,i} = ask' \cdot H(x'_i) - ask \cdot H(x_i) \; \} \end{aligned}$$

An efficient NIZKPK can be constructed for this statement using the Schnorr proof system [Sch91]. The three terms in the statement are combined with a logical "AND" operation; this can be implemented by simply sharing the Fiat-Shamir [FS87] challenge between NIZKPKs of the individual terms, as in the DLEQ construction (see Section 8.2.2.2). The third term is the most complicated to prove, since the structure of the term for each entry is uncommon. Additionally, Peng et al. [PBD07, Fig. 4] introduced a batching technique that is applicable to this term; the technique was first discussed in Section 8.2.2.3. Batching the NIZKPK significantly improves the performance, particularly in terms of size: the proof size remains constant even as the number of label-value entries increases.

Let λ_0 denote a security parameter controlling the soundness of the NIZKPK. In this setting, $\lambda_0 = 128$. Let the tag τ denote a unique encoding of the public values in the proof statement:

$$\tau = apk \| apk' \| x_1 \| x'_1 \| c_{1,i} \| c'_{1,i} \| \ldots \| x_m \| x'_m \| c_{1,m} \| c'_{1,m}$$

The prover $\mathcal{P}$ proceeds as follows:

1. Compute $e = H_1(\tau)$ where H_1 is a cryptographic hash function with $\lambda_0 \cdot m$ bits of output
2. Split e into m parts such that for each $i \in [1, m]$, the part e_i is in $[0, 2^{\lambda_0})$
3. Choose $r_1 \xleftarrow{\$} \mathbb{Z}_{p_3}$ and $r_2 \xleftarrow{\$} \mathbb{Z}_{p_3}$
4. Compute $t'_1 \leftarrow r_1 \cdot B$ and $t'_2 \leftarrow r_2 \cdot B$
5. Compute $t'_3 = r_1 \cdot \sum_{i=1}^{m}(e_i \cdot H(x'_i)) - r_2 \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i))$
6. Compute $c' \leftarrow H_2(\tau \| t'_1 \| t'_2 \| t'_3)$ where H_2 is a cryptographic hash function with λ_0 bits of output modeled by a random oracle
7. Compute $s_1 = r_1 - c' \cdot ask'$ (mod p_3) and $s_2 = r_2 - c' \cdot ask$ (mod p_3)
8. The proof is $\pi = (c', s_1, s_2)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that all values appearing in τ are valid group elements and abort otherwise
2. Compute $e = H_1(\tau)$ and split e into e_i for $i \in [1, m]$
3. Compute $t_1 \leftarrow s_1 \cdot B + c' \cdot apk'$
4. Compute $t_2 \leftarrow s_2 \cdot B + c' \cdot apk$
5. Compute $t_3 \leftarrow s_1 \cdot \sum_{i=1}^{m}(e_i \cdot H(x'_i)) - s_2 \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) + c' \cdot \sum_{i=1}^{m}(e_i \cdot (c'_{1,i} - c_{1,i}))$
6. Compute $c \leftarrow H_2(\tau \| t_1 \| t_2 \| t_3)$
7. Accept the proof if and only if $c = c'$

Section 10.7.1.2.1 proves that the NIZKPK is correct, Section 10.7.1.2.2 proves that it is zero-knowledge, and Section 10.7.1.2.3 proves that it is sound. Readers that are not interested in checking the security proofs may wish to skip to Section 10.7.1.3.

10.7.1.2.1 Proof of Correctness

If both the prover and the verifier are honest and the statement is true, then the verifier will always accept the proof. It must be shown that the inputs to the random oracle query match: $t_1 = t'_1$, $t_2 = t'_2$, and $t_3 = t'_3$. τ is computed in the same way by both parties, so it always matches.

The first task is to show that $t_1 = t'_1$:

$$
\begin{aligned}
t_1 &= s_1 \cdot B + c' \cdot apk' && \triangleright \text{ Verifier step 3} \\
&= (r_1 - c' \cdot ask') \cdot B + c' \cdot apk' && \triangleright \text{ Prover step 7} \\
&= r_1 \cdot B - c' \cdot ask' \cdot B + c' \cdot apk' && \triangleright \text{ Expand} \\
&= r_1 \cdot B - c' \cdot apk' + c' \cdot apk' && \triangleright \text{ Statement} \\
&= r_1 \cdot B = t'_1 && \triangleright \text{ Prover step 4}
\end{aligned}
$$

The second task is to show that $t_2 = t'_2$:

$$
\begin{aligned}
t_2 &= s_2 \cdot B + c' \cdot apk && \triangleright \text{ Verifier step 4} \\
&= (r_2 - c' \cdot ask) \cdot B + c' \cdot apk && \triangleright \text{ Prover step 7} \\
&= r_2 \cdot B - c' \cdot ask \cdot B + c' \cdot apk && \triangleright \text{ Expand} \\
&= r_2 \cdot B - c' \cdot apk + c' \cdot apk && \triangleright \text{ Statement} \\
&= r_2 \cdot B = t'_2 && \triangleright \text{ Prover step 4}
\end{aligned}
$$

The final task is to show that $t_3 = t'_3$:

$$
\begin{aligned}
t_3 &= s_1 \cdot \sum_{i=1}^{m} (e_i \cdot H(x'_i)) - s_2 \cdot \sum_{i=1}^{m} (e_i \cdot H(x_i)) + c' \cdot \sum_{i=1}^{m} (e_i \cdot (c'_{1,i} - c_{1,i})) && \triangleright \text{ Verifier step 5} \\
&= \sum_{i=1}^{m} (e_i \cdot (s_1 \cdot H(x'_i) - s_2 \cdot H(x_i) + c' \cdot (c'_{1,i} - c_{1,i}))) && \triangleright \text{ Merge sums} \\
&= \sum_{i=1}^{m} (e_i \cdot ((r_1 - c' \cdot ask') \cdot H(x'_i) - (r_2 - c' \cdot ask) \cdot H(x_i) + c' \cdot (c'_{1,i} - c_{1,i}))) && \triangleright \text{ Prover step 7} \\
&= \sum_{i=1}^{m} (e_i \cdot ((r_1 - c' \cdot ask') \cdot H(x'_i) - (r_2 - c' \cdot ask) \cdot H(x_i) && \triangleright \text{ Statement} \\
&\qquad + c' \cdot (ask' \cdot H(x'_i) - ask \cdot H(x_i)))) \\
&= \sum_{i=1}^{m} (e_i \cdot ((r_1 - c' \cdot ask' + c' \cdot ask') \cdot H(x'_i) && \triangleright \text{ Merge terms} \\
&\qquad - (r_2 - c' \cdot ask + c' \cdot ask) \cdot H(x_i)))
\end{aligned}
$$

$$= \sum_{i=1}^{m} (e_i \cdot (r_1 \cdot H(x'_i) - r_2 \cdot H(x_i))) \quad \triangleright \text{ Cancel}$$

$$= r_1 \cdot \sum_{i=1}^{m} (e_i \cdot H(x'_i)) - r_2 \cdot \sum_{i=1}^{m} (e_i \cdot H(x_i)) = t'_3 \quad \triangleright \text{ Prover step 5}$$

Since all terms input to the random oracle by the prover match the terms input by the verifier, $c = c'$ and the final verifier check in step step 7 will succeed.

10.7.1.2.2 Proof of Zero Knowledge

To prove that the NIZKPK is zero-knowledge, it can be shown that there exists a simulator that can produce a transcript for any given challenge that is indistinguishable from real transcripts. In other words, the NIZKPK is c-simulatable [Hen14, Def. 9] in the ROM. Constructing the simulator for this NIZKPK is straightforward, since it requires nothing more than the standard simulation approach for Schnorr-based NIZKPKs: generate the responses s_1 and s_2 at random, compute the commitments as the verifier would, and then program the random oracle accordingly. The simulator proceeds as follows when given challenge c as input:

1. Compute $e = H_1(\tau)$ and split e into e_i for $i \in [1, m]$
2. Choose $s_1 \xleftarrow{\$} \mathbb{Z}_{p_3}$ and $s_2 \xleftarrow{\$} \mathbb{Z}_{p_3}$
3. Compute $t_1 \leftarrow s_1 \cdot B + c \cdot apk'$
4. Compute $t_2 \leftarrow s_2 \cdot B + c \cdot apk$
5. Compute $t_3 \leftarrow s_1 \cdot \sum_{i=1}^{m}(e_i \cdot H(x'_i)) - s_2 \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) + c \cdot \sum_{i=1}^{m}(e_i \cdot (c'_{1,i} - c_{1,i}))$
6. Program the random oracle H_2 to output c given the input $\tau \| t_1 \| t_2 \| t_3$
7. The simulated proof is $\pi = (c, s_1, s_2)$

Note that the simulated s_1 and s_2 values perfectly match the real distribution for an honest prover, so the transcripts are indistinguishable.

10.7.1.2.3 Proof of Soundness

To prove soundness, it must be shown that if an honest verifier accepts a proof, then there exists a rewinding extractor that can recover witnesses for the statement from the prover with overwhelming probability. For this NIZKPK, an extractor exists that can recover the witness given only two challenge-response pairs for the same commitments. The extractor runs the prover in order to produce a proof $\pi = (c', s_1, s_2)$. It then rewinds the prover to the point of the query to the random oracle, reprograms the oracle to output a different challenge, and collects a second proof $\overline{\pi} = (\overline{c'}, \overline{s_1}, \overline{s_2})$. A witness can be recovered from π and $\overline{\pi}$.

Consider the random oracle input t_1. Since the verifier accepts in both instances and the complete random oracle input is the same:

$$
\begin{aligned}
s_1 \cdot B + c' \cdot apk' &= \overline{s_1} \cdot B + \overline{c'} \cdot apk' && \triangleright \text{ Verifier step 3} \\
s_1 \cdot B - \overline{s_1} \cdot B &= (\overline{c'} - c') \cdot apk' \\
\frac{s_1 - \overline{s_1}}{\overline{c'} - c'} \cdot B &= apk' \\
\frac{s_1 - \overline{s_1}}{\overline{c'} - c'} &= ask' && \triangleright \text{ Discrete logarithm}
\end{aligned}
$$

This equation shows how to compute a witness, ask', for the statement term $apk' = ask' \cdot B$. A similar computation applies to the t_2 random oracle input:

$$
\begin{aligned}
s_2 \cdot B + c' \cdot apk &= \overline{s_2} \cdot B + \overline{c'} \cdot apk && \triangleright \text{ Verifier step 4} \\
s_2 \cdot B - \overline{s_2} \cdot B &= (\overline{c'} - c') \cdot apk \\
\frac{s_2 - \overline{s_2}}{\overline{c'} - c'} \cdot B &= apk \\
\frac{s_2 - \overline{s_2}}{\overline{c'} - c'} &= ask && \triangleright \text{ Discrete logarithm}
\end{aligned}
$$

This allows recovery of the a witness, ask, for the statement term $apk = ask \cdot B$.

It remains to be shown that the extracted witnesses ask' and ask satisfy the final term of the proof statement. Firstly, consider the implication of verifier step 5 due to π being accepted:

$$
t_3 = s_1 \cdot \sum_{i=1}^{m} (e_i \cdot H(x'_i)) - s_2 \cdot \sum_{i=1}^{m} (e_i \cdot H(x_i)) + c' \cdot \sum_{i=1}^{m} (e_i \cdot (c'_{1,i} - c_{1,i}))
$$

Secondly, consider the implication of $\overline{\pi}$ being accepted:

$$t_3 = \overline{s_1} \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i')) - \overline{s_2} \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) + \overline{c'} \cdot \sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i}))$$

Next, subtract the second equation from the first equation to yield:

$$0 = (s_1 - \overline{s_1}) \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i')) - (s_2 - \overline{s_2}) \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) + (c' - \overline{c'}) \cdot \sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i}))$$

The previously established equations for the witnesses ask' and ask can be rearranged:

$$s_1 - \overline{s_1} = ask' \cdot (\overline{c'} - c')s_2 - \overline{s_2} = ask \cdot (\overline{c'} - c')$$

Substituting these values into the equation yields:

$$0 = ask' \cdot (\overline{c'} - c') \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i')) - ask \cdot (\overline{c'} - c') \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) + (c' - \overline{c'}) \cdot \sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i}))$$

The term $\overline{c'} - c'$ can now be factored out:

$$0 = (\overline{c'} - c') \cdot (ask' \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i')) - ask \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) - \sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i})))$$

Since the premise is that the rewinding extractor will choose different challenge values c' and $\overline{c'}$, it must be the case that $\overline{c'} - c' \pmod{p_3} \neq 0$. This means that the $(\overline{c'} - c')$ term can be dropped from the equation:

$$0 = ask' \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i')) - ask \cdot \sum_{i=1}^{m}(e_i \cdot H(x_i)) - \sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i}))$$

The equation can be rewritten as follows:

$$\sum_{i=1}^{m}(e_i \cdot (c_{1,i}' - c_{1,i})) = \sum_{i=1}^{m}(e_i \cdot (ask' \cdot H(x_i') - ask \cdot H(x_i)))$$

The logic of the proof now proceeds similarly to the logic in Section 8.4.1.3.2. It must be the case that every e_i was chosen independently of every $c_{1,i}$, $c_{1,i}'$, ask, ask', x_i, and x_i' value. This is due to all of these values being included as part of τ (with ask and ask' being included in the form of apk and apk'), which was used to compute every e_i value using the cryptographic hash function

H_1 in verifier step 2. The hash function ensures that the e_i values are drawn uniformly at random from $[0, 2^{\lambda_0})$. The large size of this range means that it is infeasible for a malicious prover to predict any e_i or the location of any collisions between them; this can be derived using the same logic as in Section 8.4.1.3.2. Consequently, with overwhelming probability, the equation only holds if the inner terms are equal:

$$\forall_{i=1}^{m}(c'_{1,i} - c_{1,i} = ask' \cdot H(x'_i) - ask \cdot H(x_i))$$

This is exactly the final term in the proof statement, which demonstrates that the extracted witnesses possess the correct form. This completes the proof of soundness.

10.7.1.3 Unblinding Ciphertext Table

As discussed in Section 10.1.8, the unblinding ciphertext table, $PS.U$, stores blinding key pairs that are used to blind members' identity public keys. Each blinded public key is stored in the identity table, $PS.I$, along with a reference to the blinding public key in $PS.U$ that was used to blind it. The unblinding ciphertext table includes a mapping from blinding public keys to ciphertexts containing the corresponding blinding private keys. $PS.U[bpk]$ denotes the ciphertext in the table containing the blinding private key bsk corresponding to the blinding public key bpk. The unblinding secrets, $SS.U$, contain the decrypted blinding private keys: $SS.U[bpk]$ denotes the decrypted $PS.U[bpk]$ value stored in the private set. As discussed in Section 10.7, $PS.U$ makes use of an anchor key pair, similarly to the construction for the confidential label-value store described in Section 10.7.1.1. The implementation for the unblinding ciphertext table is simpler than the confidential label-value store: no elliptic curve groups are used, each blinding private key in the table is encrypted to the anchor public key using only an SBRAKEM encapsulation, and there is no per-entry freshness marker.

Figure 10.3 depicts the structure of the unblinding ciphertext table. In summary, the corresponding portion of the public state contains the following data:

- $PS.U.T$: an SBRAKEM encapsulation of $SS.U.ask$ to the set of invitation public keys in $PS.L$.
- $PS.U.apk$: the anchor public key in $\mathbb{G}_2$.
- $PS.U.f$: a bit indicating if ask is fresh.
- For the entry with label bpk:
  - $PS.U[bpk]$: an SBRAKEM encapsulation of the blinding private key bsk corresponding to bpk, transferred to ask.

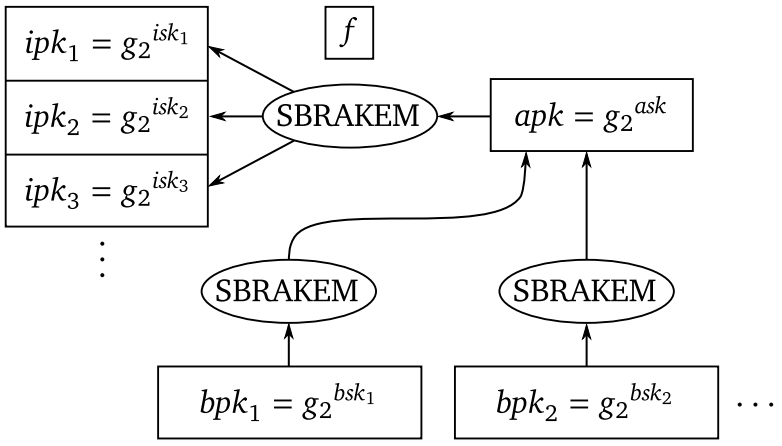


Figure 10.3 THE UNBLINDING CIPHERTEXT TABLE STRUCTURE WHEN USING $\text{BRAKEM}^{\text{DDL}}_{\star}$. The anchor private key *ask* is transferred to the invitation public keys in *PS.L* using $\text{BRAKEM}^{\text{DDL}}_{\star}$. The entries in the table are public keys of the form $bpk_i = g_2{}^{bsk_i}$. Each blinding private key bsk_i is transferred to the anchor public key using $\text{BRAKEM}^{\text{DDL}}_{\star}$. f indicates the anchor key pair's freshness. (Ref: 382)

The following data is stored in the private state:

- *SS.U.ask*: the anchor private key.
- For the entry with label *bpk*:
  - *SS.U*[*bpk*]: the blinding private key *bsk* corresponding to *bpk*.

The following sections describe how to implement the LVSM operations and the decryption function for *PS.U*.

10.7.1.3.1 Initialize

The **LVSM.*U*.Initialize** operation proceeds as follows:

Execute CKeyGen with:
- Output pk^*: store in *PS.U.apk*
- Output sk^*: store in *SS.U.ask*.

$PS.U.f \leftarrow 1$.

10.7.1.3.2 Refresh Invitations

The **LVSM.U.RefreshInvites** operation proceeds as follows:

Let R denote the set of invitation public keys stored in $PS.L$.
if (performer context) {
    $T \leftarrow$ SBRAKEM.Encapsulate$(R, PS.U.apk, SS.U.ask)$.
    Write T to the commit.
} **else** {
    Read T from the commit.
    if (SBRAKEM.Verify$(R, PS.U.apk, T) \neq 1$) { Abort. }
}
$PS.U.T \leftarrow T$.

10.7.1.3.3 Refresh Encryption

The **LVSM.U.RefreshEncryption** operation proceeds as follows:

$apk' \leftarrow PS.U.apk$.
if $(SS \neq \perp)$ { $ask' \leftarrow SS.U.ask$. }
Execute LVSM.U.Initialize. ▹ Update anchor key pair
Execute LVSM.U.RefreshInvites. ▹ Transfer anchor private key to invitations
Let R' = KC.BroadcastKeys$(PS.G)$. ▹ Transfer anchor private key to members
if (performer context) {
    $T' \leftarrow$ SBRAKEM.Encapsulate$(R', PS.U.apk, SS.U.ask)$.
    Write T' to the commit.
} **else if** $(SS = \perp)$ {
    if (SBRAKEM.Verify$(R', PS.U.apk, T') \neq 1$) { Abort. }
} **else** {
    Find $r \in R'$ such that $r \rightarrow sk$ is in $SS.Q$.
    Let j represent the position of r in R'.
    $SS.U.ask \leftarrow$ SBRAKEM.Decapsulate$(R', j, sk, PS.U.apk, T')$.
    if $(SS.U.ask = \perp)$ { Abort. }
}
for each (bpk such that $PS.U[bpk]$ exists, sorted) { ▹ Re-encrypt private keys
    if (performer context) { $bsk \leftarrow SS.U[bpk]$. }
    else { $bsk \leftarrow \perp$. }

Execute LVSM.U.Set with:
- Parameter bpk: set to bpk
- Parameter bsk: set to bsk.

}
$PS.U.f \leftarrow 1$.

10.7.1.3.4 Mark Stale

The **LVSM.U.MarkStale** operation proceeds as follows:

$PS.U.f \leftarrow 0$.

10.7.1.3.5 Set

The **LVSM.U.Set** operation proceeds as follows:

Parameters: bpk, a public key; and bsk, either a private key to store or $\perp$.

---

if (bpk is not valid) { Abort. }
if (performer context) {
  if ((bpk, bsk) is not a valid keypair) { Abort. }
  $T \leftarrow$ SBRAKEM.Encapsulate($\{PS.U.apk\}, bpk, bsk$).
  Write T to the commit.
} **else** {
  Read T from the commit.
  if ($SS = \perp$) {
    if (SBRAKEM.Verify($\{PS.U.apk\}, bpk, T$) $\neq 1$) { Abort. }
  } **else** {
    $bsk \leftarrow$ SBRAKEM.Decapsulate($\{PS.U.apk\}, 1, SS.U.ask, bpk, T$).
    if ($bsk = \perp$) { Abort. }
    $SS.U[bpk] \leftarrow bsk$.
  }
}
$PS.U[bpk] \leftarrow T$.

10.7.1.3.6 Delete

The **LVSM.U.Delete** operation proceeds as follows:

Parameters: bpk, a public key.

---

if ($PS.U[bpk]$ exists) { Delete $PS.U[bpk]$. }
if ($SS \neq \perp$) {
    if ($SS.U[bpk]$ exists) { Delete $SS.U[bpk]$. }
}

10.7.1.3.7 Decrypt

The **LVSM.U.Decrypt**$(PS, SS, id, isk) \rightarrow SS'$ function proceeds as follows:

Let R denote the set of invitation public keys stored in $PS.L$.
Let j refer to the invitation in R with identifier id.
$SS' \leftarrow SS$.
$SS'.U.ask \leftarrow$ SBRAKEM.Decapsulate$(R, j, isk, PS.U.apk, PS.U.T)$.
if ($SS'.U.ask = \perp$) { **return** $\perp$. }
for each (bpk such that $PS.U[bpk]$ exists) {
    $bsk \leftarrow$ SBRAKEM.Decapsulate$(\{PS.U.apk\}, 1, SS'.U.ask, bpk, PS.U[bpk])$.
    if ($bsk = \perp$) { Abort. }
    $SS.U[bpk] \leftarrow bsk$.
}
return SS'.

10.8 Authentication NIZKPKs

As discussed in Section 10.1.6, a Safehouse session has configurable authentication ephemerality and deniability settings. The session may either use long-term identities, or only ephemeral identities. In terms of deniability, the session can be configured in a "non-repudiable", "offline deniable", or "strongly deniable" mode.

When long-term identities are enabled for a Safehouse session, members must authenticate themselves to the other members. An overview of this mechanism was given in Section 10.1.9. Members authenticate themselves by producing a NIZKPK in three situations:

- New members authenticate to existing members when they join the session.
- When existing members come online, they authenticate to any new members that joined the session while they were offline.
- When an existing member changes their identity for post-compromise security purposes, they authenticate the change.

The proof statements for the authentication NIZKPKs vary widely based on the situation and the group mode.

In all cases, the authentication NIZKPK demonstrates knowledge of a set of discrete logarithms arranged in a specific logical expression. An *identity witness* contains a set of private keys that satisfies the proof statement. The proof statement is always of the form:

$$PK\{(\text{discrete logarithms}) : \text{honest} \vee \text{forged}\}$$

The "honest" term is a conjunction of discrete logarithms, potentially including discrete logarithm equalities. The exact term depends on the which of the three aforementioned scenarios the proof is for. The "forged" term depends only on the deniability mode of the Safehouse session. It allows forgers to produce the proofs for transcripts, thereby providing plausible deniability.

The following algorithm summarizes what is proven as part of the "honest" term, where knowledge of the discrete logarithm of a blinded public key is proven with respect to the corresponding blinding public key:

```
if (the member is newly joining the session) {
    Prove knowledge of an invitation private key corresponding to a layaway in PS.L.
    if (long-term identities are enabled) {
        Prove knowledge of the private key for the new blinded public key.
        if (the invitation is an individual invitation) {
            Prove that the private key for the blinded public key in the layaway
            ↪ matches the private key for the new blinded public key.
        }
    } else if (the invitation is an individual invitation) {
        Prove knowledge of the private key for the blinded public key in the layaway.
```

```
    }
} else if (long-term identities are enabled) {
    if (the member is authenticating to newly joined members) {
        Prove knowledge of the private key for the member's entry in PS.I.
        if (switching to a new public key for blinding) {
            Prove that the private key for the member's entry in PS.I
            ↪ matches the private key for the new blinded public key.
        }
    } else if (the member is changing their identity) {
        Prove knowledge of the private key for the new blinded public key.
    }
}
```

One potentially unexpected aspect of the "honest" algorithm is that when authenticating during an identity changing operation, the member does not need to prove knowledge of their previous blinded public key. This is because this operation will be digitally signed using the member's personal private key, as with most other MLS-layer operations. Since all of the recipients of the NIZKPK in this scenario have already authenticated the prover, they have already bound the prover's identity to the personal key used to sign the operation. This is not true in the other situations, which both involve new members that lack these established associations.

Another notable aspect of the "honest" algorithm is that it allows existing members to blind their identity public key using a new blinding key pair in $PS.U$ when authenticating to new members. This is an important feature, because it allows the number of entries in $PS.U$ to decrease over time as existing members converge on a smaller set of blinding bases. The NIZKPK requires proving that the discrete logarithm of the previous blinded public key with respect to the previous blinding public key matches the discrete logarithm of the new blinded public key with respect to the new blinding public key. This proof provides assurance to the new members that the "identified" identity public key they previously assigned to the prover is associated with the newly "authenticated" identity public key.

During normal operation, identity witnesses contain only the private keys required to satisfy the "honest" term of the proof statement, and the "forged" term is ignored. When a Safehouse session is configured to provide deniability, a forger can use an identity witness containing private keys that satisfy the "forged" term to produce authentication NIZKPKs. The following algorithm summarizes what is proven as part of the "forged" term, where knowledge of the discrete logarithm of a blinded public key is proven with respect to the corresponding blinding public key:

```
if (long-term identities are enabled && the session is not in non-repudiable mode) {
    Initialize an empty sequence R_1.
    for each (personal key for an existing or newly joining member) {
        Append the personal public key to R_1.
    }
    Initialize an empty sequence R_2.
    if (the session is in strongly deniable mode) {
        for each (blinded public key for an existing or newly joining member) {
            Append the blinded public key to R_2.
        }
    }
    for each (index i in R_1) {
        if (the session is in strongly deniable mode) {
            Prove knowledge of the private key for the i^th entry in either R_1 or R_2.
        } else {
            Prove knowledge of the private key for the i^th entry in R_1.
        }
    }
}
```

When configured in "offline" or "strong" deniability modes,[21] Safehouse makes it possible to plausibly deny that an identity public key is associated with a member in a session, because the transcript of that session could be forged.

If a Safehouse session is not in the "non-repudiable" mode, then any offline forger can forge a transcript given only the identity public keys of the parties to include in the transcript. An offline forger can simulate the behavior of all of the parties and use their personal private keys to forge authentication proofs. Since the offline forger generates all of the personal private keys for the simulated parties, it is able to use an identity witness containing all of the private keys for the entries in R_1.

When a session is in the "strong" deniability mode, then it is also possible for a misinformant to forge proofs that are sent to the interactive judge, as long as the misinformant knows the identity private keys for all of the members that are reporting to the judge. The misinformant can simulate a session containing the members reporting to the judge and an arbitrary set of simulated members that are not actually present (as long as the misinformant knows the identity public keys of the members to simulate). For each member in the session, the misinformant's identity witness contains the private key for the associated entry in R_2 if the member is under

[21] ^ The relevant forms of deniability are defined in Section 2.4.

the control of the judge, or for the entry in R_1 if they are simulating the member.[22] This allows the misinformant to satisfy the "forged" term of any authentication NIZKPK by using the identity private keys of members reporting to the judge and the personal private keys of members that are being simulated.

Importantly, the structure of the proof statement allows a misinformant to produce an authentication NIZKPK for a newly joining simulated member even if they do not posses any invitation private keys, which may be generated by the judge, since knowledge of invitation private keys is part of the "honest" term. This method of achieving strong deniability is very similar to the approach used by DAKEZ (see Section 5.4), except extended into the group setting.

During normal operation, every honest member in a Safehouse knows that incoming authentication NIZKPKs cannot be produced by identity witnesses using the "forged" term, because the honest member knows (but cannot prove to outsiders) that no other entities possess their personal private key or identity private key. Consequently, any valid authentication proof received by the member must have been produced using an identity witness for the "honest" term.

While the algorithms above are sufficient to derive the proof statements, explicit proof statements are listed for the "honest" term in Section 10.8.1 and for the "forged" term in Section 10.8.2. These proof statements are denoted using the following notational conventions, with the corresponding proof statements given in Camenisch-Stadler notation [CS97]:[23]

$$\mathrm{DL}\{\alpha : (h, x)\} := PK\{(\alpha) : x = h^{\alpha}\}$$
$$\mathrm{DLEQ}\{\alpha : (h_1, x_1) \approx (h_2, x_2)\} := PK\{(\alpha) : x_1 = {h_1}^{\alpha} \wedge x_2 = {h_2}^{\alpha}\}$$

The statements are given with respect to a generic group $\mathbb{G}$ with generator g and order q, since the actual group will be the key space of the developer's chosen BRAKEM scheme. Sections 10.8.4 and 10.8.5 describe two NIZKPKs constructions for the statements.

10.8.1 Honest Proof Term

This section uses the following notation for variables in the various proof statements:

[22] The misinformant cannot use the personal private key of members controlled by the judge, because the judge might provide a personal public key to use, preventing the misinformant from learning the personal private key. The judge has no control over the other simulated parties, so the misinformant can freely generate personal key pairs for them.

[23] The notation for DLEQ is copied from Section 8.2.2.2.

- (ipk, isk) denotes the invitation key pair for the invitation used by a newly joining member. ipk is the invitation public key stored in $PS.L$ for the layaway that is specified by the member upon joining the session.
- (pk, sk) denotes the identity key pair for the prover.
- $\overline{pk}$ denotes the blinded public key for the prover—the value from $PS.I$ for existing members, or the new value for joining members.
- (bpk, bsk) denotes the blinding key pair used to produce $\overline{pk}$ from pk using the Blind function (see Section 10.1.8).

When a new member joins the session using a bearer invitation and the session is configured to use only ephemeral identities, the "honest" term is:

$$\mathrm{DL}\{isk : (g, ipk)\}$$

When a new member joins the session using a bearer invitation and the session is configured to use long-term identities, the "honest" term is:

$$\mathrm{DL}\{isk : (g, ipk)\} \wedge \mathrm{DL}\{sk : (bpk, \overline{pk})\}$$

When a new member joins the session using an individual invitation, the session is configured to use long-term identities, and the associated layaway contains the blinded public key $\overline{tpk}$ that has been blinded using the blinding public key $btpk$, the "honest" term is:

$$\mathrm{DL}\{isk : (g, ipk)\} \wedge \mathrm{DLEQ}\{sk : (bpk, \overline{pk}) \approx (btpk, \overline{tpk})\}$$

When an existing member authenticates to new members and retains its blinded public key, the "honest" term is:

$$\mathrm{DL}\{sk : (bpk, \overline{pk})\}$$

When an existing member authenticates to new members while switching to a new blinded public key $\overline{pk}'$ that has been blinded using the blinding public key bpk', the "honest" term is:

$$\mathrm{DLEQ}\{sk : (bpk, \overline{pk}) \approx (bpk', \overline{pk}')\}$$

When an existing member changes to a new identity, the "honest" term is:

$$\mathrm{DL}\{sk : (bpk, \overline{pk})\}$$

10.8.2 Forged Proof Term

When the session is configured in non-repudiable mode, the "forged" term is:

False

When the session is configured in offline deniable mode and R_1 denotes the set of personal public keys for all existing and newly joining members, the "forged" term is:

$$\bigwedge_{pk_i \in R_1} \mathrm{DL}\{sk_i : (g, pk_i)\}$$

When the session is configured in strongly deniable mode, R_1 denotes a sequence of the personal public keys for all existing and newly joining members, R_2 denotes the blinded public keys for the same sequence of members, B denotes the blinding public keys used to blind the corresponding entries of R_2, and the sequences contain n elements accessed using square brackets, the "forged" term is:

$$\bigwedge_{i=1}^{n} \Big(\mathrm{DL}\{sk_{1,i} : (g, R_1[i])\} \vee \mathrm{DL}\{sk_{2,i} : (B[i], R_2[i])\} \Big)$$

10.8.3 Interface

The authentication NIZKPKs are exposed to the other operational layers in the form of the following two functions:

- **Auth.Prove**$(ipk, bpk, \overline{pk}, btpk, \overline{tpk}, R_1, B, R_2, m, w) \rightarrow \pi$.
- **Auth.Verify**$(ipk, bpk, \overline{pk}, btpk, \overline{tpk}, R_1, B, R_2, m, \pi) \rightarrow b$.

Some of the arguments may be omitted by setting them to the special symbol $\perp$. R_1 is a sequence of personal public keys (or $\perp$), B is a sequence of blinding public keys (or $\perp$), and R_2 is a sequence of blinded public keys. $B \neq \perp$ if and only if $R_2 \neq \perp$. The three sequences must all have the same length, except for sequences that are $\perp$. R_1, B, and R_2 will always contain at least one element if they are not set to $\perp$, because they must contain public keys for every existing and newly joining member; the sequences will always contain a public key corresponding to the member producing the proof (or they will be set to $\perp$). w denotes the identity witness. The output of Auth.Verify is $b = 1$ if and only if the proof π is valid. The value m is *associated data*, which is an opaque blob that is unambiguously concatenated into the Fiat-Shamir challenge for the NIZKPK; this ensures

that the NIZKPK is bound to a particular context.[24] The functions choose the "honest" term (see Section 10.8.1) and the "forged" term (see Section 10.8.2) to use based on the arguments according to the following algorithm:

if $(ipk \neq \perp$ **&&** $bpk = \perp$ **&&** $btpk = \perp)$ {
    honest := DL$\{isk : (g, ipk)\}$
} **else if** $(ipk = \perp$ **&&** $bpk \neq \perp$ **&&** $btpk = \perp)$ {
    honest := DL$\{sk : (bpk, \overline{pk})\}$
} **else if** $(ipk \neq \perp$ **&&** $bpk \neq \perp$ **&&** $btpk = \perp)$ {
    honest := DL$\{isk : (g, ipk)\} \wedge$ DL$\{sk : (bpk, \overline{pk})\}$
} **else if** $(ipk \neq \perp$ **&&** $bpk \neq \perp$ **&&** $btpk \neq \perp)$ {
    honest := DL$\{isk : (g, ipk)\} \wedge$ DLEQ$\{sk : (bpk, \overline{pk}) \approx (btpk, \overline{tpk})\}$
} **else if** $(ipk = \perp$ **&&** $bpk \neq \perp$ **&&** $btpk \neq \perp)$ {
    honest := DLEQ$\{sk : (bpk, \overline{pk}) \approx (bpk', \overline{pk}')\}$
} **else** {
    Abort.
}
if $(R_1 = \perp$ **&&** $R_2 = \perp)$ {
    forged := False
} **else if** $(R_1 \neq \perp$ **&&** $R_2 = \perp)$ {
    forged := $\bigwedge_{pk_i \in R_1}$ DL$\{sk_i : (g, pk_i)\}$
} **else if** $(R_1 \neq \perp$ **&&** $R_2 \neq \perp)$ {
    forged := $\bigwedge_{i=1}^{n}$(DL$\{sk_{1,i} : (g, R_1[i])\} \vee$ DL$\{sk_{2,i} : (B[i], R_2[i])\}$)
} **else** {
    Abort.
}
stmt := honest $\vee$ forged.
return stmt.

[24] For an example of associated data in NIZKPKs produced with the Fiat-Shamir transform [FS87] and how it typically interacts with the surrounding system, see the discussion in Section 5.7 of the associated data for the DAKEs presented in Chapter 5.

10.8.4 Construction: Schnorr Compiler

There are five different possible forms[25] for the "honest" term, as shown in Section 10.8.1, and three different possible forms for the "forged" term, as shown in Section 10.8.2. This is a total of 15 possible proof statements. This section describes an approach for implementing the NIZKPKs for these proof statements.

The most straightforward way to implement NIZKPKs for all of the proof statements is to use the Schnorr proof system [Sch91] with an appropriate NIZKPK compiler. All of the individual components are DL or DLEQ proof statements. DLEQ statements appear in components of the "honest" term when a new member joins using an individual invitation or when an existing member switches to a new blinded public key. Constructions for DL and DLEQ statements using the Schnorr proof system were presented in Sections 8.2.2.1 and 8.2.2.2, respectively. While these constructions are given in the context of the $\text{BRAKEM}_{\star}^{\text{DDL}}$ group setting, they also work in other group settings, such as the elliptic curve group for $\text{BRAKEM}^{\text{ZK}}$.

Combining NIZKPKs for DL and DLEQ proof statements into an overall authentication NIZKPK can be accomplished by using compilers for logical "AND" and "OR" operations. Combining two Schnorr-based NIZKPKs with an "AND" operation is trivial: the two random oracle queries are instead combined into a single random oracle query, and the challenges produced by the two random oracles are instead output from the combined random oracle. This is essentially equivalent to "simultaneously" performing both NIZKPKs as normal, yielding the same effect as a logical "AND" operation. Combining two Schnorr-based NIZKPKs with an "OR" operation can be done using the technique described by Cramer et al. [CDS94], which was previously used in Chapter 5 to build a ring signature scheme (see Section 5.3.3) that forms the basis of the new DAKEZ, ZDH, and XZDH DAKEs. In general, the compiler combines two Schnorr-based NIZKPKs as follows:

1. Denote the original proofs as $\pi_1 = (c_1, s_1)$ and $\pi_2 = (c_2, s_2)$, each consisting of the Fiat-Shamir *challenge*, c_i, and the *response*, s_i. Originally, the verifier would compute $c_1 = H_1(t_1)$ and $c_2 = H_2(t_2)$, where H_1 and H_2 are the random oracles for the original NIZKPKs and t_1 and t_2 are the *commitments*.

2. The Fiat-Shamir challenge in the newly unified proof is denoted by c'. The challenges used in the original proofs must satisfy the constraint $c' = c_1 + c_2$. The prover specifies c' and c_1, thereby implicitly specifying c_2. The unified proof becomes $\pi = (c', c_1, s_1, s_2)$.

[25] The form for an identity change is the same as the form for an existing member authenticating to new members while retaining its blinded public key.

3. The verifier computes $c_2 = c' - c_1$ and derives the original queries t_1 and t_2 from (c_1, s_1) and (c_2, s_2) as before. It then computes $c = H(t_1 \| t_2)$, where H is the random oracle.
4. The verifier checks that $c = c'$.

Without loss of generality, let the prover's witness satisfy the i^{th} original proof, but not the j^{th} original proof, where $j = 3 - i$. The prover can produce the combined proof as follows:

1. Choose a random c_j value.
2. Run the zero-knowledge simulator for the j^{th} original proof with challenge c_j, producing t_j and s_j.
3. Run the honest proving algorithm for the i^{th} original proof normally up to the point of the random oracle query, producing t_i.
4. Compute $c' = H(t_1 \| t_2)$.
5. Compute $c_i = c' - c_j$.
6. Complete the honest proving algorithm for the i^{th} original proof with challenge c_i, producing s_i.
7. The proof is $\pi = (c', c_1, s_1, s_2)$.

Together, the "AND" and "OR" compilers can produce NIZKPKs for all 15 possible proof statements. Security proofs for these NIZKPKs follow directly from the security proofs for the underlying DL and DLEQ NIZKPKs and the security proofs for the two compilers.

10.8.4.1 BDL: Batch Discrete Logarithm

When the session is configured in the "offline deniable" mode, the "forged" term becomes a simple conjunction of DL NIZKPKs, as discussed in Section 10.8.2. The NIZKPK compilation technique described in Section 10.8.4 can produce a correct and secure proof system for this statement without issue: the "AND" compiler for the Schnorr proofs will result in a proof that contains one hash (the challenge) and one group element for each public key in R_1. An alternative approach that provides better performance is to use a batched proof system similar to the BDLEQ scheme presented in Section 8.2.2.3; the resulting proof requires only the challenge hash and

one group element. Recall that the "forged" term in the "offline deniable" mode can be written as a conjunction of n DL statements:

$$\bigwedge_{i=1}^{n} \mathrm{DL}\{sk_i : (g, pk_i)\}$$

The batched proof system for this statement is called Batch Discrete Logarithm (BDL), and it can be constructed using the same technique as the BDLEQ construction. The prover $\mathcal{P}$ produces the NIZKPK, with security parameter λ_0, as follows:

1. Choose $r \xleftarrow{\$} \mathbb{Z}_q$
2. Compute $t' \leftarrow g^r$
3. Compute $c' \leftarrow H_2(pk_1 \| \ldots \| pk_n \| t')$ where H_2 is a cryptographic hash function with λ_0 bits of output modeled by a random oracle
4. Compute $e = H_1(c')$ where H_1 is a cryptographic hash function with $\lambda_0 \cdot n$ bits of output
5. Split e into n parts such that for each $i \in [1, n]$, the part e_i is in $[0, 2^{\lambda_0})$
6. Compute $s \leftarrow r - \Sigma_{i=1}^{n}(e_i \cdot sk_i) \pmod{q}$
7. The proof is $\pi = (c', s)$

The verifier $\mathcal{V}$ checks the proof as follows:

1. Verify that $pk_1, \ldots, pk_n \in \mathbb{G}$ and abort otherwise
2. Compute $e = H_1(c')$ and split e into e_i for $i \in [1, n]$
3. Compute $t \leftarrow g^s \cdot \prod_{i=1}^{n}\left(pk_i^{e_i}\right)$
4. Compute $c \leftarrow H_2(pk_1 \| \ldots \| pk_n \| t)$
5. Accept the proof if and only if $c = c'$

This previously known BDL construction is ultimately based on the "small exponent" technique introduced by Bellare et al. [BGR98, §3.3] in the context of batched verification of digital signatures. Hoshino et al. [HAK01, §2.2] were the first to informally apply the technique to

zero-knowledge proofs.[26] Peng et al. [PBD07] formalized the batch verification techniques and proved the security of these schemes. Henry [Hen14, Fig. 3.3] extended the formalization and presented the most accessible description of the construction. The security of the BDL NIZKPK described above follows from the security proofs given by Henry [Hen14, Th. 3.16] for the interactive version of the proof system. The aforementioned NIZKPK was produced by applying the Fiat-Shamir transform [FS87] to the interactive protocol. An optimization is also included in the NIZKPK presented above: instead of deriving all n of the e_i terms from the commitment t' and including these terms in the proof, the e_i terms are instead derived from the random oracle output. This optimization significantly reduces the size of the proof compared to the standard result from the Fiat-Shamir transform, without affecting the applicability of the security proofs.

The BDL proof system can be used to prove the "forged" term when in "offline deniable" mode. In this case, the "forged" term is still combined with the "honest" term using the "AND" compiler: the random oracle queries in the proofs for the two terms are combined into a single random oracle query, producing a unified proof. The BDL proof is not used in other modes; in particular, the "forged" term in the "strongly deniable" mode must still be derived by appropriately combining applications of the "AND" and "OR" compilers.

10.8.4.2 Example

For an example of the approach for constructing NIZKPKs described in Section 10.8.4, consider the most complex scenario: a new member joining a session, configured to use long-term identities and in "strongly deniable" mode, using an individual invitation. Combining the "honest" term from Section 10.8.1 with the "forged" term from Section 10.8.2 yields the following combined proof statement:

$$\left(\mathrm{DL}\{isk : (g, ipk)\} \wedge \mathrm{DLEQ}\{sk : (bpk, \overline{pk}) \approx (btpk, \overline{tpk})\}\right)$$
$$\vee\left(\bigwedge_{i=1}^{n}\left(\mathrm{DL}\{sk_{1,i} : (g, R_1[i])\} \vee \mathrm{DL}\{sk_{2,i} : (B[i], R_2[i])\}\right)\right)$$

[26] The proof system introduced by Hoshino et al. is applicable to a generalized family of proof statements. Their construction applies to the BDL statement when the system's parameter is set to $k = 1$.

When written out in full Camenisch-Stadler notation [CS97], the statement becomes:

$$PK\Bigg\{(isk, sk, sk_{1,1}, sk_{2,1}, \ldots, sk_{1,n}, sk_{2,n}) : \\ \left(ipk = g^{isk} \wedge \overline{pk} = bpk^{sk} \wedge \overline{tpk} = btpk^{sk}\right) \vee \\ \left(\bigwedge_{i=1}^{n} \left(R_1[i] = g^{sk_{1,i}} \vee R_2[i] = B[i]^{sk_{2,i}}\right)\right)\Bigg\}$$

After applying the proof compilers to the original NIZKPKs, the proof becomes:

$$\pi = (c', c_1, s_{1,1}, s_{1,2}, c_{2,1,1}, s_{2,1,1}, s_{2,1,2}, \ldots, c_{2,n,1}, s_{2,n,1}, s_{2,n,2})$$

The challenges for the constituent NIZKPKs are related in the following way, modulo the group order q:

- $c' = c_1 + c_2 \pmod{q}$.
- $\forall_{i=1}^{n}(c_2 = c_{2,i,1} + c_{2,i,2}) \pmod{q}$.

Note that c_2 and $c_{2,i,2}$ for $1 \leq i \leq n$ are not explicitly included in π—the verifier can derive them from the other terms in π based on the required relationships. The original proofs are represented in the combined NIZKPK by the following terms:

- DL$\{isk : (g, ipk)\}$ is represented by $(c_1, s_{1,1})$.
- DLEQ$\{sk : (bpk, \overline{pk}) \approx (btpk, \overline{tpk})\}$ is represented by $(c_1, s_{1,2})$.
- DL$\{sk_{1,i} : (g, R_1[i])\}$ is represented by $(c_{2,i,1}, s_{2,i,1})$.
- DL$\{sk_{2,i} : (B[i], R_2[i])\}$ is represented by $(c_{2,i,2}, s_{2,i,2})$.

A prover with a witness satisfying the "honest" term would compute the proof as follows:

1. Choose a random c_2.
2. For $i = 1$ to n, choose a random $c_{2,i,1}$ and compute $c_{2,i,2} = c_2 - c_{2,i,1} \pmod{q}$.

3. Simulate the proofs in the "forged" term using the DL zero-knowledge simulator with the assigned challenges. For each entry $i \in [1, n]$, this produces commitments $t_{2,i,1}$ and $t_{2,i,2}$ and responses $s_{2,i,1}$ and $s_{2,i,2}$.
4. Produce honest commitments $t_{1,1}$ and $t_{1,2}$ for the DL and DLEQ proofs in the "honest" term.
5. Query the random oracle to derive c'.
6. Compute $c_1 = c' - c_2$.
7. Complete the proofs for the "honest" term, producing responses $s_{1,1}$ and $s_{1,2}$.
8. Output the proof π.

In contrast, a prover with a witness satisfying the "forged" term would compute the proof as follows:

1. Choose a random c_1.
2. Simulate the proofs in the "honest" term using the DL and DLEQ zero-knowledge simulators with challenge c_1, producing commitments $t_{1,1}$ and $t_{1,2}$, and responses $s_{1,1}$ and $s_{1,2}$.
3. For $i = 1$ to n:
   a) Let $b_i \in \{1, 2\}$ denote the index of the statement in the i^{th} "forged" term for which the prover knows a witness, and let $\overline{b_i} = 3 - b_i$.
   b) Produce an honest commitment t_{2,i,b_i} for the DL proof.
   c) Choose a random $c_{2,i,\overline{b_i}}$.
   d) Simulate the other DL proof for the i^{th} term, producing commitment $t_{2,i,\overline{b_i}}$ and response $s_{2,i,\overline{b_i}}$.
4. Query the random oracle to derive c'.
5. Compute $c_2 = c' - c_1$.
6. For $i = 1$ to n:
   a) Compute $c_{2,i,b_i} = c_2 - c_{2,i,\overline{b_i}}$.
   b) Complete the honest proof for the term, producing response s_{2,i,b_i}.
7. Output the proof π.

This example demonstrated the proof compiler approach applied to the most complicated of the 15 proof statements. The approach can be applied to the other proof statements to derive similar NIZKPKs. Proof statements for the "offline deniable" mode can use the BDL proof system introduced in Section 10.8.4.1 for the "forged" term instead of repeated applications of the "AND" compiler. Deriving the other NIZKPKs using the compilers is straightforward.

10.8.5 Construction: Borromean Ring Signatures

Section 10.8.4 described a construction for the authentication NIZKPKs using standard Schnorr proof compilers for "AND" and "OR" expressions. This approach is relatively simple and easy to implement using common cryptographic libraries. However, using a more sophisticated proving technique, it is possible to significantly decrease the size of the proofs without introducing any new security assumptions. This section provides an overview of a more efficient approach based on Borromean ring signatures [MP15]. Borromean ring signatures are a technique for constructing efficient NIZKPKs for proof statements consisting of terms that can be implemented using Schnorr-based proof systems. Prior to this work, Borromean ring signatures were only applied to proof statements consisting of DL proofs combined by "AND" and "OR" operators. The contribution of this section is to generalize Borromean ring signatures so that they can contain DLEQ terms, allowing them to be applied to the authentication proof statements required by Safehouse.

The main advantage of Borromean ring signatures over the approach in Section 10.8.4 is that a Borromean ring signature requires only one challenge (i.e., hash output) to be transferred in the proof. This results in a significantly reduced proof size, especially when using the "forged" term for the "strongly deniable" mode introduced in Section 10.8.2: the approach in Section 10.8.4 produces proofs with $2+n$ hash values, where n denotes the number of members, while Borromean ring signatures require only 1 hash value in the proof.

In a standard NIZKPK created using the Fiat-Shamir transform [FS87], the commitments are part of the random oracle query, and the output of the random oracle (the challenge) is used by the verifier to derive the commitments. In the Borromean ring signature framework, the random oracle output and the challenge used by the verifier are decoupled. In other words, the value output by the random oracle when queried with the commitments is not necessarily the challenge value used by the verifier to derive the commitments (and thus also by the prover to produce the response). Figure 10.4 depicts this situation in the form of a directed graph representation. The vertices represent challenge values, and the edge represents a NIZKPK using the Fiat-Shamir transform for some proof statement. Let T_0 denote the complete random oracle query in the original proof, including all of the necessary commitments. The random oracle is

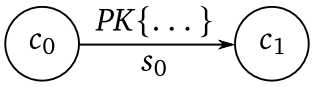


Figure 10.4 AN EDGE IN A BORROMEAN RING SIGNATURE GRAPH. This is an incomplete graph that shows the fundamental building block of a Borromean ring signature. The directed edge represents a NIZKPK. The output of the random oracle query is c_1. The response s_0 is computed using the challenge value c_0. A prover can compute c_1 in two possible ways: by using a witness for the proof statement to execute the honest proving algorithm, or by using advance knowledge of c_0 to execute the zero-knowledge simulator with c_0 as the challenge. (Refs: 400 and 401)

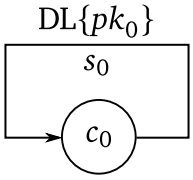


Figure 10.5 A BORROMEAN RING SIGNATURE GRAPH FOR $\mathrm{DL}\{pk_0\}$. This proof is equivalent to a standard Schnorr NIZKPK of a discrete logarithm (Refs: 401 [a] [b])

queried as in the original proof, producing $c_1 = H(T_0)$ (the target vertex of the edge). However, the challenge used by the prover to complete the proof is c_0 (the source vertex of the edge) instead of c_1. When using a standard Schnorr proof of knowledge of a discrete logarithm x_0 with commitment $T_0 = g^{r_0}$ for the edge, the prover would compute the challenge $c_1 = H(T_0)$ and the response $s_0 = r_0 + c_0 \cdot x_0$. Figure 10.4 does not depict a complete Borromean ring signature—only the fundamental building block. The critical observation is that c_1 can be computed in two ways: given a witness for the proof statement, the honest NIZKPK proving algorithm can be used to derive c_1 from the random oracle output; or given c_0, a c-simulatable [Hen14, Def. 9] zero-knowledge simulator for the NIZKPK can be used to derive s_0 and c_1. In terms of the graph representation, this means that it is possible to compute the challenge for the target of an edge given a witness for the edge, or the challenge for the source of the edge.

Borromean ring signatures become useful when the associated graph representation contains cycles, also known as *causal loops*. Consider the graph structure shown in Figure 10.5. The Borromean ring signature produced by the prover for this graph is $\pi = (c_0, s_0)$. The graph contains a trivial cycle with the reflexive edge from c_0 to itself. $\mathrm{DL}\{pk_0\}$ denotes a proof statement demonstrating knowledge of the discrete logarithm of pk_0 with respect to some base; the base is irrelevant to the discussion in this section. This reflexive edge means that the output of the random oracle for this NIZKPK is also used to compute and verify the response, s_0. In other words, the graph in Figure 10.5 corresponds to a Borromean ring signature that is exactly equivalent to a standard Schnorr NIZKPK of a discrete logarithm. The verifier knows that the target of the edge,

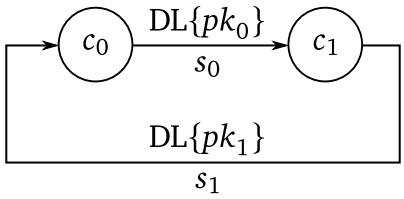


Figure 10.6 A BORROMEAN RING SIGNATURE GRAPH FOR $\mathrm{DL}\{pk_0\} \vee \mathrm{DL}\{pk_1\}$. The verifier can deduce that the prover knows the witness for at least one of the two edges. (Ref: 402)

c_0, could not have been computed by the prover by using knowledge of the source of the edge, c_0, because these values are the same, and no prover can find a value with such a relationship due to the properties of the random oracle. Therefore, the verifier can conclude that the target of the edge, c_0, must have been computed using a witness for the edge (i.e., the prover knows the discrete logarithm of pk_0).

Borromean ring signatures can be used to prove more interesting statements by using more complex graphs. Consider the graph structure shown in Figure 10.6. The Borromean ring signature produced by the prover for this graph is $\pi = (c_0, s_0, s_1)$. The verifier checks the proof as follows:

1. Compute the commitment T_0 using the challenge c_0 and the response s_0.
2. Compute the random oracle output $c_1 = H(T_0)$.
3. Compute the commitment T_1 using the challenge c_1 and the response s_1.
4. Compute the random oracle output $c'_0 = H(T_1)$.
5. Accept the proof if and only if $c'_0 = c_0$.

Since the graph contains a cycle, it is not possible for the prover to satisfy the verification process by running the zero-knowledge simulator for both DL statements. This means that the verifier can conclude that at least one edge in the cycle was produced by using a witness for the edge. There are two possibilities:

1. The prover used a witness sk_0 to derive c_1 and s_0 using the honest proving algorithm, and then used the zero-knowledge simulator with c_1 to produce c_0 and s_1.
2. The prover used a witness sk_1 to derive c_0 and s_1 using the honest proving algorithm, and then used the zero-knowledge simulator with c_0 to produce c_1 and s_0.

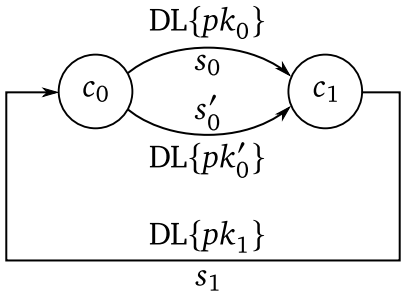


Figure 10.7 A BORROMEAN RING SIGNATURE GRAPH FOR $(\text{DL}\{pk_0\} \wedge \text{DL}\{pk'_0\}) \vee \text{DL}\{pk_1\}$. c_1 is the output of a random oracle query with commitments for both $\text{DL}\{pk_0\}$ and $\text{DL}\{pk'_0\}$. (Refs: 403, 404 a b, and 405)

Therefore, a Borromean ring signature for this graph corresponds to an "OR" proof of the two DL NIZKPKs; the prover must know at least one of the two witnesses. Note that the resulting proof differs from the equivalent proof produced using the technique from Cramer et al. [CDS94] that was discussed in Section 10.8.4.

In addition to implementing logical "OR" operators in proof statements, Borromean ring signatures can also be used to implement the "AND" operator. Accomplishing this requires a small extension to the aforementioned technique. In the previously discussed graphs, all vertices have an in-degree of at most 1. When a vertex in a graph for a Borromean ring signature has a larger in-degree, this denotes that it is produced as the output of a random oracle call where the commitments for *all* of the incoming edges are included in the random oracle query. For example, consider the graph shown in Figure 10.7. The Borromean ring signature for this graph is $\pi = (c_0, s_0, s'_0, s_1)$. If T_0 denotes the commitment for the NIZKPK for $\text{DL}\{pk_0\}$ and T'_0 denotes the commitment for the NIZKPK for $\text{DL}\{pk'_0\}$, then $c_1 = H(T_0 \| T'_0)$ (i.e., c_1 is the output of a random oracle call that unambiguously contains both T_0 and T'_0 in the query). There are two possible ways for a prover to compute c_1:

1. Given witnesses sk_0 and sk'_0 for $\text{DL}\{pk_0\}$ and $\text{DL}\{pk'_0\}$, the prover can use the honest proving algorithm to create commitments T_0 and T'_0, compute c_1, and then produce the responses s_0 and s'_0.

2. Given a value of c_0, the prover can use the zero-knowledge simulator for $\text{DL}\{pk_0\}$ and $\text{DL}\{pk'_0\}$ to choose random responses s_0 and s'_0, derive the expected commitments T_0 and T'_0, and then compute c_1.

In other words, the effect of edges can be generalized for vertices in the graph with in-degrees larger than 1: for each incoming edge, the prover must either know the witness for the corresponding proof statement, or it must know the value of the source vertex. When multiple

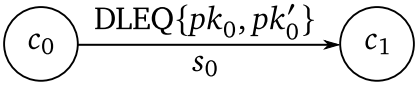


Figure 10.8 AN EDGE FOR A DLEQ PROOF IN A BORROMEAN RING SIGNATURE GRAPH. (Ref: 404)

incoming edges have the same source vertex, as in Figure 10.7, this has the effect of requiring the prover to know the value for the source vertex (c_0), or the witnesses for all of the edges. Consequently, subgraphs of this shape can be used to implement the "AND" operator in the overall proof. Figure 10.7 depicts the shape as part of a larger cycle, thereby implementing a Borromean ring signature for the more complex statement $(\text{DL}\{pk_0\} \wedge \text{DL}\{pk'_0\}) \vee \text{DL}\{pk_1\}$.

Finally, although Maxwell and Poelstra [MP15] describe Borromean ring signatures only in the context of DL NIZKPKs, edges in the graph may correspond to more complicated proof statements. For example, Figure 10.8 depicts an edge that corresponds to a DLEQ proof statement; $\text{DLEQ}\{pk_0, pk'_0\}$ denotes a NIZKPK demonstrating that the discrete logarithms of pk_0 and pk'_0 are equal with respect to some bases (which are omitted for clarity). The standard DLEQ NIZKPK built using the Schnorr proof system, which was presented in Section 8.2.2.2, can be used as part of this Borromean ring signature without issue: the proving algorithm and c-simulatable zero-knowledge simulator can be used in exactly the same way as for edges corresponding to DL NIZKPKs. Internally, the commitment T_0 for the DLEQ NIZKPK is actually a commitment to two group elements. Using the notation from the NIZKPK in Section 8.2.2.2, $T_0 = t_1 \| t_2$, where the verifier computes $t_1 \leftarrow {h_1}^{s_0} \cdot {x_1}^{c_0}$ and $t_2 \leftarrow {h_2}^{s_0} \cdot {x_2}^{c_0}$.

10.8.5.1 Example

Given the building blocks described in Section 10.8.5, it is possible to implement Borromean ring signatures for all of the authentication proof statements discussed in Section 10.8 by formulating appropriate graphs. Figure 10.9 depicts the graph for a Borromean ring signature that corresponds to the most complex authentication proof statement used by Safehouse: a new member joining a session, configured to use long-term identities and in "strongly deniable" mode, using an individual invitation. This is the same as the example used to illustrate the output of the Schnorr "AND" and "OR" proof compilers in Section 10.8.4.2. Recall that the proof statement is:

$$\left(\text{DL}\{isk : (g, ipk)\} \wedge \text{DLEQ}\{sk : (bpk, \overline{pk}) \approx (btpk, \overline{tpk})\}\right)$$
$$\vee \left(\bigwedge_{i=1}^{n} \left(\text{DL}\{sk_{1,i} : (g, R_1[i])\} \vee \text{DL}\{sk_{2,i} : (B[i], R_2[i])\}\right)\right)$$

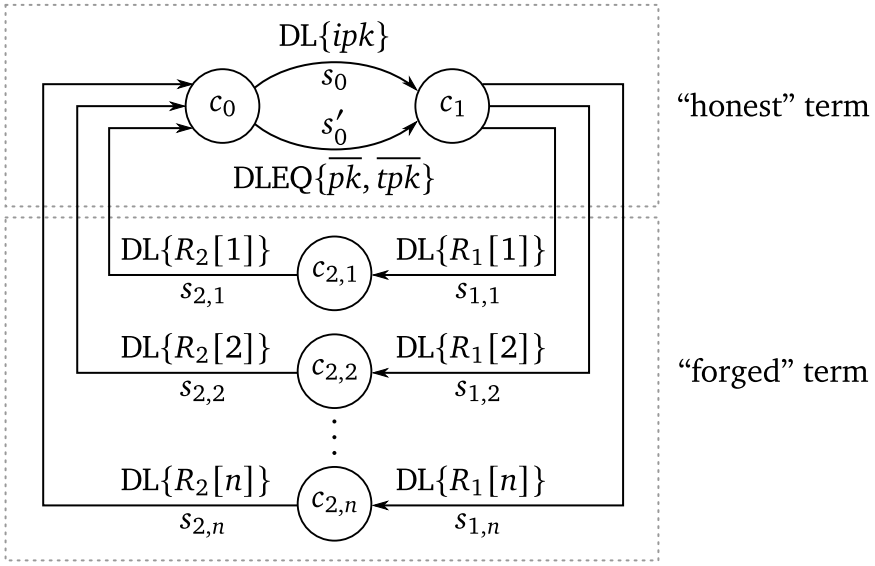


Figure 10.9 A BORROMEAN RING SIGNATURE GRAPH FOR THE MOST COMPLEX AUTHENTICATION PROOF. A proof for this graph demonstrates knowledge of witnesses for the two NIZKPKs in the "honest" term, or for at least one DL for each of the n indices in the "forged" term. (Ref: 404)

The overall statement is an "honest" term and a "forged" term combined with a logical "OR" operator, so the graph for the Borromean ring signature consists of an overall cycle with two subgraphs—this forces the prover to know all of the witnesses for one of the two subgraphs. The subgraph for the "honest" term uses the "AND" technique from Figure 10.7 to combine the DL and DLEQ proof statements. The "forged" term is a conjunction of n disjunctions with 2 terms each, so the corresponding subgraph consists of n 2-step paths from the c_1 vertex to the c_0 vertex. The final Borromean ring signature for the statement is:

$$\pi = (c_0, s_0, s'_0, s_{1,1}, s_{2,1}, \ldots, s_{1,n}, s_{2,n})$$

Borromean ring signatures are called "signatures" because they can be bound to a particular message m by including m in all random oracle queries. As discussed in Section 10.8.3, Safehouse authentication NIZKPKs must be bound to a particular context using "associated data". When using Borromean ring signatures to implement authentication NIZKPKs, the associated data is the "message" that is included in the random oracle queries. The prototype implementation of Safehouse uses the construction previously described in Section 10.8.4; formulation of the Borromean ring signature graphs for all 15 possible statements and an implementation of the construction is left as future work.

10.9 Interface Functions

Developers interact with a Safehouse implementation using its public interface. This interface can be divided into two parts: a set of *interface functions*, and the MLS layer. Interface functions allow the developer to interact with the current group state in prescribed ways. Interface functions do not modify the group state, and they do not produce commits. In contrast, the MLS layer consists of operations (as defined in Section 10.1.1) that modify the group state and produce commits to send to the server. MLS-layer operations are performed to update the group state, whereas interface functions are called to make use of the currently established group state.

This section describes the interface functions exposed by Safehouse. Section 10.9.1 describes the function that derives application-specific keys from the group key, Section 10.9.2 describes functions for serializing and deserializing group states, and Section 10.9.3 presents functions for authenticating payloads. Section 10.10 presents the rest of Safehouse's public API: the MLS layer.

10.9.1 Group Key Derivation

The primary purpose of Safehouse is to establish and maintain a shared secret key that group members can use to communicate. The group key, $SS.K$, is this shared secret key. However, since $SS.K$ is used internally in the Safehouse protocol (e.g., in the KGM.RefreshKeys operation described in Section 10.5.8 and the KGM.RatchetGroupKey operation described in Section 10.5.9), directly exposing it to the developer is problematic. If an application built using Safehouse were given direct access to the group key, then a security proof would need to account for the application's use of the group key. It is only possible to produce a composable security proof for Safehouse if the application's view is restricted to one-way functions of the group key; the one-wayness property can be relied upon to prevent the behavior of the overall application from undermining key secrecy. The design of Safehouse avoids revealing the group key in order to facilitate the future development of composable security proofs and verification tools.

The following interface function is provided as part of Safehouse's API:

- **SharedKey**$(PS, SS, \Phi, \ell) \rightarrow K$: produces an ℓ-bit shared secret key K associated with the group state (PS, SS) and the domain separation string Φ.

The SharedKey function deterministically produces a key K based on the tuple $(SS.K, \Phi, \ell)$. In other words, all members of the session with the same public state PS will derive the same key if they provide the same values for Φ and ℓ. The function's output is computationally independent

of outputs for different arguments: it should not be possible to efficiently distinguish between outputs for different choices of Φ and ℓ without access to $SS.K$.

For any group state (PS, SS) produced by a member's honest execution of the Safehouse protocol, any PPT adversary has negligible advantage in the following game:

1. The challenger sends PS to the adversary.
2. The adversary is given oracle access for SharedKey$(PS, SS, \cdot, \cdot)$. The adversary may perform a polynomially bounded number of SharedKey oracle calls and other operations.
3. The adversary outputs strings Φ_0 and Φ_1, and non-negative integers ℓ_0 and ℓ_1.
4. The challenger computes $K_0 \leftarrow$ SharedKey(PS, SS, Φ_0, ℓ_0).
5. The challenger computes $K_1 \leftarrow$ SharedKey(PS, SS, Φ_1, ℓ_1).
6. If $\ell_0 < \ell_1$ then the challenger chooses $K' \xleftarrow{\$} \{0,1\}^{\ell_1 - \ell_0}$ and sets $K_0 \leftarrow K_0 \| K'$.
7. If $\ell_1 < \ell_0$ then the challenger chooses $K' \xleftarrow{\$} \{0,1\}^{\ell_0 - \ell_1}$ and sets $K_1 \leftarrow K_1 \| K'$.
8. The challenger chooses $b \xleftarrow{\$} \{0,1\}$.
9. If $b = 0$ then the challenger sets $k_0 \leftarrow K_0$ and $k_1 \leftarrow K_1$.
10. If $b = 1$ then the challenger sets $k_0 \leftarrow K_1$ and $k_1 \leftarrow K_0$.
11. The challenger sends (k_0, k_1) to the adversary.
12. The adversary may perform a polynomially bounded number of SharedKey oracle calls and other operations.
13. The adversary outputs a guess b'.
14. The adversary wins if $b' = b$ and it never queried the oracle for SharedKey(PS, SS, Φ_0, ℓ_0) or SharedKey(PS, SS, Φ_1, ℓ_1).

Note that this security game implicitly requires that the SharedKey implementation produces keys that are computationally indistinguishable from random strings. If a SharedKey implementation did not provide this property, then an adversary could choose to output $\ell_0 = 0$ and $\ell_1 > 0$; the game would then reduce to distinguishing SharedKey outputs from random strings. It is also important to note that it is easy for an adversary to win this game against an implementation for

which SharedKey(PS, SS, Φ, ℓ_0) is a prefix (or any public function) of SharedKey(PS, SS, Φ, ℓ_1) for $\ell_0 < \ell_1$.

In practice, it is easy to implement SharedKey using an XOF that generates random and unrelated outputs, such as cSHAKE128 [KCP16]. The XOF can be given a unique encoding of $(SS.K, \Phi, \ell)$ as the preimage and asked to produce ℓ bits of output. Since the adversary does not know $SS.K$, it can not directly compute the SharedKey outputs—it must guess the order by examining the output streams. However, the cSHAKE128 outputs are unrelated [KCP16, §8.2.2] and indistinguishable from random [Dwo15, §A.2], so the outputs do not reveal any useful information to the adversary.

10.9.2 State Serialization

The ability to serialize and deserialize the public state (defined in Section 10.1.2) is needed for both transmission to new members over the network and for ensuring that the state hash, $PS.H$, can prevent forked group states from being maliciously "merged". Most of the difficult work involved in serializing or deserializing the state is done by the KC subsystem, which, as discussed in Section 10.6, provides a structure-aware efficient serialization mechanism for the key graph, $PS.G$, the public key map, $PS.P$, the superfluous edges, $PS.S$, and the identity table, $PS.I$. Knowledge of special key graph structure does not help to optimize serialization of the other items in the public state, so this process is the same regardless of the developer's choice of KC implementation. The exact details of how to serialize the remaining contents of PS are straightforward implementation concerns that are outside the scope of this chapter.[27] However, it is important to define the serialization interface so that it can be clearly referenced in MLS-layer operations. Additionally, as noted in Section 10.6, it is critically important that the serialization process produces a lossless encoding; otherwise, a malicious server may be able to undermine the insider security guarantees. Moreover, the serialization function must produce a unique encoding for a given group state, but it is acceptable for the deserialization function to (losslessly) decode multiple alternative representations into the same group state.[28]

[27] ^ $PS.H$, $PS.X$, $PS.M$, $PS.A$, $PS.E$, $PS.U$, and $PS.L$ can all be efficiently serialized in the obvious way, using fixed-length fields, length-prefixed strings, and an efficient integer encoding method.

[28] ^ State serialization is used for three purposes in Safehouse: to encode the group state in the input to hash functions as part of operations, to save the group state in persistent storage, and to transmit the current public state to newly joining members. In the first and second cases, since the serialization function produces a unique encoding, all honest members will serialize the same group state in the same manner, so there is no possibility for the adversary to produce different serializations for the same group state. It is possible for a malicious server to send different serializations of the current group state to different joining members, but this has no effect: since the encoding is lossless, all members will recover the same group state to proceed with the joining process. The actual serialized binary blob sent to newly joining members is not used for any other purpose.

The following interface functions are provided as part of Safehouse's API:

- **SerializePS**$(PS) \rightarrow D$: a function that accepts a public state PS and losslessly encodes it as a unique opaque binary blob D. Internally, SerializePS calls KC.Serialize (see Section 10.6) to serialize the relevant portions of PS.
- **DeserializePS**$(D) \rightarrow PS$: a function that accepts a binary blob D and losslessly decodes it as a public state. Multiple values of D may produce the same output. Internally, DeserializePS calls KC.Deserialize to invert KC.Serialize's output.
- **SerializeSS**$(SS) \rightarrow D$: a function that accepts a private state SS and losslessly encodes it as a unique opaque binary blob D.
- **DeserializeSS**$(D) \rightarrow SS$: a function that accepts a binary blob D and losslessly decodes it as a private state SS. Multiple values of D may produce the same output.

SerializeSS and DeserializeSS are only used by the developer to store a member's private state in a storage device; these functions are not needed internally by Safehouse.

10.9.3 Commit and Payload Signing

Internally, Safehouse ensures that all commits are digitally signed by a member in the group. To avoid interfering with the deniability of the transcript, commits are usually signed by the performing member's personal key, personal(U_{perf}), instead of a long-term key.[29] The exception is when an invitation is issued and the session is configured to use anonymous invitations (see Section 10.1.6). When an invitation is issued anonymously, the commit that adds the corresponding layaway to $PS.L$ is signed using a key known to all members in the session, KC.SharedKey$(PS.G)$, rather than the personal key of the performer. All other commits are signed by the performing member's personal key. For KC implementations in which a shared key does not exist, the group cannot be set into anonymous invitation mode.[30] In all cases, digitally signing a commit using a k-node requires the definition of a digital signature scheme that operates in the same key space as the developer's chosen BRAKEM construction. It is necessary to use the

[29] The deniability system discussed in Section 10.8 ensures that personal keys can only be provably bound to long-term identities by members. Although the signed commits provably link operations to a particular member, that member cannot be linked to a long-term identity when the session is in a deniable mode.

[30] Alternatively, the commit could be signed using a ring signature over KC.BroadcastKeys$(PS.G)$. The prototype implementation of Safehouse simply disallows this configuration instead of supporting it with a dedicated ring signature scheme. This is not a significant concern because a shared key exists for most KC schemes.

public key associated with the k-node as the signature verification key in order to support public verifiability—the server, with no access to private keys, must be able to verify the signature in order to decide if it should reject the commit.

The signature scheme for the BRAKEM key space consists of two functions:

- **Sig**$(pk, sk, m) \rightarrow \sigma$: produces a signature σ on message m using signing key sk corresponding to verifying key pk.
- **SVerif**$(pk, m, \sigma) \rightarrow b$: outputs $b = 1$ if σ is a valid signature on message m for verifying key pk, and $b = 0$ otherwise.

This is the same scheme definition that is required as a parameter to the authenticated TKLL implementation, which was described in Section 9.4. An implementation of this scheme is implicitly provided to Safehouse alongside the chosen BRAKEM construction. Commits are internally signed using the Sig function with the signing key corresponding to personal(U_{perf}) (for most commits) or corresponding to KC.SharedKey$(PS.G)$ (for anonymous invitations). These signatures are internally verified using the SVerif function upon receipt.

As mentioned in Section 9.4.2, implementing the required digital signature scheme is trivial for both BRAKEM$^{\text{DDL}}_{\star}$ and BRAKEM$^{\text{ZK}}$. For BRAKEM$^{\text{DDL}}_{\star}$, the Schnorr signature scheme [Sch91] can be used: this is essentially a DL NIZKPK, which is used widely throughout Safehouse, that also includes m in the random oracle query. For BRAKEM$^{\text{ZK}}$, the EdDSA signature scheme can be used with the subgroup on the Jubjub elliptic curve that is used for the key space (see Section 8.5.2).

In general, reusing keys across cryptosystems is not guaranteed to preserve security. It is important to ensure that the digital signature scheme and the BRAKEM scheme can use the same keys while remaining secure. The Schnorr signature scheme uses the same NIZKPK construction as all of the proofs in BRAKEM$^{\text{DDL}}_{\star}$, and thus it is trivial to extend the security proofs for BRAKEM$^{\text{DDL}}_{\star}$ to include the digital signature scheme. The EdDSA signature scheme is ultimately built from the same ECDH key exchange protocol as the ElGamal cryptosystem in the BRAKEM$^{\text{ZK}}$ construction. While the security of this combination would not be as trivial to prove as the BRAKEM$^{\text{DDL}}_{\star}$ combination, the digital signature scheme is still fundamentally compatible with BRAKEM$^{\text{ZK}}$.

One of the design objectives of Safehouse, as discussed in Chapter 7, is providing optional sender authentication for payloads. This is accomplished by exposing functionality to the developer that signs payloads in the same way that commits are signed. The following interface functions are provided as part of Safehouse's API:

- **AuthMember**$(PS, SS, aid, m) \rightarrow \sigma$: signs a message m as the member with agent ID aid in the session with group state (PS, SS), producing a signature σ. Internally, the function sets $pk = \text{personal}(aid)$ and locates a mapping $pk \rightarrow sk$ in $SS.Q$. If no such mapping is found, the function returns $\sigma = \perp$. Otherwise, it returns $\sigma = \text{Sig}(pk, sk, m)$.

- **VerifyMember**$(PS, aid, m, \sigma) \rightarrow b$: verifies that σ is a valid signature for the message m by the member with agent ID aid in the session with public state PS. The function returns $b = 1$ if and only if the signature is valid. Internally, it computes $b = \text{SVerif}(\text{personal}(aid), m, \sigma)$.

- **AuthAny**$(PS, SS, m) \rightarrow \sigma$: signs a message m using a signing key known to all members in the session with group state (PS, SS), producing a signature σ. This function is the same as AuthMember, except that it internally sets $pk = \text{KC.SharedKey}(PS.G)$. If the KC scheme does not provide a shared key, then the function returns $\sigma = \perp$.

- **VerifyAny**$(PS, m, \sigma) \rightarrow b$: verifies that σ is valid signature for the message m produced by a member in the session with public state PS. This function is the same as VerifyMember, except that it sets pk in the same way as AuthAny.

The developer can use the four functions above to authenticate payloads. AuthMember and VerifyMember can be used to authenticate that a specific member signed a message. Alternatively, AuthAny and VerifyAny can be used to authenticate that *some* member signed a message, without revealing the specific member's identifier. As with the anonymity preservation feature, while AuthAny does not specifically reveal the member that produced the signature, the developer must take additional steps to ensure real-world anonymity (e.g., avoiding a leak of the author's identity through network metadata). A benefit of using AuthAny is that the resulting signature is verifiable by the server, so the server can reject invalid messages without storing or forwarding them. In cases where this functionality is not needed, the developer can simply use authenticated encryption or a MAC using a key derived from the group key (see Section 10.9.1).

10.10 Message Layer Security

This section defines the Safehouse operations in the MLS layer. As discussed in Sections 10.1.3 and 10.9, these operations are the portion of the Safehouse API that allows the developer to update the group state.

MLS-layer operations modify the group state as part of an update context. Performer contexts generate a commit when they are finished. This commit is relayed by the server to the other

members. The other members use receiver contexts to process the commit and update their own group state. The following functions are exposed to the developer for managing update contexts:

- **MLS.PerformerContext**$(PS, SS) \rightarrow ctx$: initializes a performer context ctx given the initial group state (PS, SS).
- **MLS.ReceiverContext**$(PS, SS, T) \rightarrow ctx$: initializes a receiver context ctx given the initial group state (PS, SS) and the commit T. New members (when receiving a joining commit) and the server (always) use $SS = \perp$.
- **MLS.Intermediate**$(ctx) \rightarrow (PS, SS)$: returns the intermediate group state between operations. This can be used by the group policy to determine if subsequent operations should be permitted.
- **MLS.Finish**$(ctx) \rightarrow T$: finalizes the operations performed in the performer context ctx, producing a commit T. The commit is digitally signed using the Sig function, as described in Section 10.9.3.
- **MLS.Apply**$(ctx) \rightarrow (PS, SS)$: finalizes the operations that were performed in the receiver context ctx, producing a new group state (PS, SS) if the commit was valid, or $\perp$ if the commit was invalid. Validating the commit includes checking that the digital signature is valid using the SVerif function, as described in Section 10.9.3. For the server, applying a valid commit always produces $PS \neq \perp$ and $SS = \perp$.

The update context internally contains the intermediate group state and the partially assembled (for performer contexts) or partially consumed (for receiver contexts) commit. As MLS operations are performed, the intermediate group state and commit are updated. Operations from other layers, such as the KGM (see Section 10.5), KC (see Section 10.6), and LVSM (see Section 10.7) layers, are all implicitly given access to the intermediate group state and commit. Before applying an MLS-layer operation, the developer's group policy can use MLS.Intermediate to determine whether or not the operation is permitted. Similarly, the group policy can consult the history of MLS-layer operations before the call to MLS.Finish or MLS.Apply in order to evaluate the transaction as a whole. This configurable behavior was described in Section 10.1.6.

The MLS-layer update context also includes a CBRAKEM context, as defined in Section 10.2, called *bctx*. MLS.PerformerContext calls $bctx \leftarrow$ CBRAKEM.PerformerContext() to initialize the context. MLS.Finish calls $T' \leftarrow$ CBRAKEM.Finish($bctx$) to produce a CBRAKEM commit T' and includes T' in the MLS commit, T. MLS.ReceiverContext immediately recovers T' from the given commit T and calls $bctx \leftarrow$ CBRAKEM.ReceiverContext(T') to initialize the context. MLS.Apply returns $\perp$ if CBRAKEM.Apply($bctx$) returns 0.

The remainder of this section describes all of the operations that can be applied to MLS-layer update contexts. Unless otherwise noted, these operations are exposed to the developer.

10.10.1 Internal Helpers

10.10.1.1 Complete Operation

The **MLS.CompleteOperation** operation is an internal helper that is not exposed to the developer. It is called at the end of most operations in the MLS layer in order to ensure that the resulting group state after the operation is safe and that the group key has been updated. As discussed in Section 10.1.3, this is a mandatory postcondition for MLS-layer operations. The state hash $PS.H$ is updated using a domain-separated cryptographic hash function, H, before being used by KGM.RefreshKeys to derive the new group key.[31] This process ensures that forked sessions can never be maliciously "merged".

Parameters: D, a description of the MLS-layer operation that was just performed, expressed as a sequence of arbitrary elements; and b, a bit indicating whether or not the old group key should be required to derive the new group key.

Let D' represent a unique and lossless encoding of the contents of D.
$PS.H \leftarrow H(PS.H \| U_{\text{perf}} \| b \| \text{SerializePS}(PS) \| D')$.
$R \leftarrow \text{KC.BroadcastKeys}(PS.G)$.
Execute KGM.RefreshKeys with:

- Parameter R: set to R
- Parameter b: set to b
- Parameter *bctx*: set to *bctx*
- Output *bctx*: store in *bctx*.

The new state hash $PS.H$ that is produced by MLS.CompleteOperation is derived from the previous public state (represented by the previous state hash), a description of the operation that was performed (represented by U_{perf}, b, and D), and the new public state. Since the previous and new public states are included in the hash input, it is not possible for the adversary to cause honest members to arrive at different public states without forking the group. However, it is important to guarantee that honest members also arrive at the same public state through the

[31] The domain-separated hash function H is used in a similar role as distinct KDFs elsewhere in the protocol. Although the underlying implementation is likely the same, it would be misleading to call H a KDF because $PS.H$ is not a key.

same sequence of steps. If a malicious server colludes with a malicious member, they might theoretically be able to perform different operations that result in the same public state, causing honest members to experience a different sequence of operations without forking the group. To prevent the possibility of such attacks, D must include all of the public information about the MLS-layer operation that was performed, with the exception of information that appears in the new public state (since the new public state is always used in the derivation of $PS.H$).

10.10.1.2 Replace Keys

The **MLS.ReplaceKeys** operation is an internal helper that is not exposed to the developer. This operation replaces the key pairs for a subset of the k-nodes known to U_{perf}. The procedure is easy to implement correctly by combining other operations: an "adversary" is temporarily added to the group state with knowledge of the member's keys, and it is then "evicted" from the session.

Parameters: R, a set of k-nodes such that $R \subseteq \text{keyset}_{PS.G}(U_{\text{perf}})$.

Choose an agent ID for the "adversary", *aid*, that does not appear in $PS.G$.
Add a u-node for *aid* to $PS.G$.
Add a personal key for *aid* to $PS.G$.
for each $(r \in R)$ {
    Add $aid \rightarrow r$ to $PS.S$.
}
Execute KGM.Evict with:
- Parameter U: set to $\{aid\}$
- Parameter *bctx*: set to *bctx*
- Output *bctx*: store in *bctx*.

Execute KGM.DiscardMember with:
- Parameter U: set to *aid*.

10.10.1.3 Deniability Sets

The **DeniabilitySet**$(PS, U) \rightarrow (R_1, B, R_2)$ function is an internal helper that is not exposed to the developer. It determines the sets of keys to use for authentication proofs in order to support the session's deniability mode. DeniabilitySet returns a set of personal keys R_1, a set of blinding public keys B, and a set of blinded public keys R_2. Section 10.8.2 discusses how the R_1, B, and R_2 sets are used in the authentication proof. If $U = \perp$, then the sets contain entries for all members

in the session. Otherwise, the sets only contain entries for members in the session that are also found in the set U. The function proceeds as follows:

Set R_1 to $\perp$ if PS is in "non-repudiable" mode, or $\emptyset$ otherwise.
Set B and R_2 to $\perp$ if PS is not in "strongly deniable" mode, or $\emptyset$ otherwise.
for each $(aid \rightarrow (\overline{pk}, bpk)$ in $PS.I$, sorted by $aid)$ {
    if $(U \neq \perp$ **&&** $aid \notin U)$ { **continue** }
    if (the session is not in "non-repudiable" mode) { Append personal(aid) to R_1 }
    if (the session is in "strongly deniable" mode) {
        Append $\underline{bpk}$ to B.
        Append $\overline{pk}$ to R_2.
    }
}
return (R_1, B, R_2).

10.10.2 Initialize

The **MLS.Initialize** operation initializes the group state for a new Safehouse session. It is the only operation for which the initial public state PS is $\perp$.

Parameters: M, the initial developer-supplied group mode.

---

if $(PS \neq \perp)$ { Abort. }
Initialize PS with the correct structure (see Section 10.1.2).
Set $PS.X$ to an empty set.
Set $PS.S$ to an empty graph.
$PS.M \leftarrow M$.
Set $PS.A$ to an empty map.
Set $PS.L$ to an empty set.
Execute LVSM.E.Initialize.
Execute LVSM.U.Initialize.
if (performer context) {
    Initialize SS with the correct structure (see Section 10.1.2).
    Set $SS.K$ to a random key.
    $(PS.G, PS.P, SS.Q, T') \leftarrow$ KC.InitGraphPerform(U_{perf}).
    Write T' to the commit.
} **else** {

```
    Read T′ from the commit.
    (PS.G, PS.P) ← KC.InitGraphApply(U_perf, T′).
}
if (long-term identities are enabled) {
    if (performer context) {
        (bpk, bsk) ← BRAKEM.KeyGen().
        pk̄ ← Blind(pk, bpk, bsk).
        Write bpk and pk̄ to the commit.
    } else {
        Read bpk and pk̄ from the commit.
        bsk ← ⊥.
    }
}
Execute LVSM.U.Set with:
  ▸ Parameter bpk: set to bpk
  ▸ Parameter bsk: set to bsk.
```

10.10.3 Invitations

There are three MLS-layer operations that are used to modify the set of layaways in $PS.L$. All three operations are affected by the "invitation anonymity" configuration for the setting, discussed in Section 10.1.6. When the InviteAnonymity value returned by SessionConfig($PS.M$) is false, the commits produced by these three functions are digitally signed by personal(U_{perf}) as usual. When InviteAnonymity is true, the commits are signed by the private key associated with a k-node known to all group members instead. Specifically, the k-node returned by KC.SharedKey($PS.G$) (see Section 10.6) is used to sign the commits.

Over time as layaways in $PS.L$ are removed (either due to being used, expiring, or retraction) and added, invitation identifiers may be reused. This approach avoids the need to store the historical invitation count in $PS.L$ or to use long and randomly generated identifiers, but it raises the possibility that an agent may try to join the group using an invitation identifier that no longer corresponds to the expected invitation key pair. Such an attempt to join the group will eventually fail when the joining member is unable to produce the required authenticating NIZKPK. However, a "fast fail" mechanism can be introduced into Safehouse deployments to avoid wasting time attempting to join the group when the effort is doomed to fail: the newly joining member can provide a hash of the expected invitation public key to the server alongside the invitation identifier

when arranging to join the session. This mechanism enables the server to quickly reject clients using outdated invitations prior to starting the "mass join" procedure.

10.10.3.1 Invite Bearer

The **MLS.InviteBearer** operation issues a new bearer invitation to join the session.

Parameters: e, an approximate expiry date for the invitation; u the maximum number of times that the invitation can be used to join the group, or $\perp$ for unlimited uses.
Output: I, either the invitation for secure out-of-band transmission, or $\perp$.

Let iid be the smallest positive integer that does not appear as an identifier in $PS.L$.
Execute CKeyGen with:
- Output pk^*: store in ipk
- Output sk^*: store in isk.

Add $(iid, ipk, e, \perp, \perp, u)$ to $PS.L$.
Execute LVSM.E.RefreshInvites.
Execute LVSM.U.RefreshInvites.
Execute MLS.CompleteOperation with:
- Parameter D: set to ("InviteBearer", e, u)
- Parameter b: set to 1.

if (performer context) { **return** $I = (iid, ipk, isk, bpk, \overline{pk})$. }
else { **return** $I = \perp$. }

10.10.3.2 Invite Individual

The **MLS.InviteIndividual** operation issues a new individual invitation to join the session. The performing member provides the unblinded identity public key as a parameter, while receiving agents leave the parameter empty. The identity public key is blinded as part of the operation.

Parameters: e, an approximate expiry date for the invitation; pk, either an unblinded identity public key, or $\perp$.
Output: I, either the invitation for secure out-of-band transmission, or $\perp$.

---

if (long-term identities are not enabled) { Abort. }
Let iid be the smallest positive integer that does not appear as an identifier in $PS.L$.
Execute CKeyGen with:

▸ Output pk^*: store in ipk
▸ Output sk^*: store in isk.
if (performer context) {
  $(\underline{bpk}, bsk) \leftarrow$ BRAKEM.KeyGen().
  $\overline{\underline{pk}} \leftarrow$ Blind($pk, \underline{bpk}, bsk$).
  Write bpk and $\overline{\underline{pk}}$ to the commit.
} **else** {
  Read bpk and $\overline{\underline{pk}}$ from the commit.
  $bsk \leftarrow \perp$.
}
Add $(iid, ipk, e, bpk, \overline{\underline{pk}}, 1)$ to $PS.L$.
Execute LVSM.E.RefreshInvites.
Execute LVSM.U.RefreshInvites.
Execute MLS.CompleteOperation with:
▸ Parameter D: set to ("InviteIndividual", e)
▸ Parameter b: set to 1.
if (performer context) { **return** $I = (iid, ipk, isk, bpk, \overline{\underline{pk}})$. }
else { **return** $I = \perp$. }

10.10.3.3 Retract Invitation

The **MLS.RetractInvitation** operation removes a layaway from $PS.L$, preventing future use of the associated invitation.

Parameters: iid, the identifier for the invitation to retract.

Remove the layaway with the identifier iid from $PS.L$.
if (no such layaway was found) { Abort. }
Execute LVSM.E.MarkStale.
Execute LVSM.E.RefreshInvites.
Execute LVSM.U.MarkStale.
Execute LVSM.U.RefreshInvites.
Execute MLS.CompleteOperation with:
▸ Parameter D: set to ("RetractInvitation", iid)
▸ Parameter b: set to 1.

10.10.4 Mass Join

The **MLS.MassJoin** operation adds one or more new members to the session; these new members are called *joining members* during the joining process. The operation is parameterized by a joining commit (see Section 10.6) produced by the KC.PrepareJoin protocol. One of the agents that executed KC.PrepareJoin acts as U_{perf} on behalf of all of the joining members. U_{perf} digitally signs the commit using personal(U_{perf}) as it exists at the *end* of the operation, rather than at the start (as is the case for all other MLS-layer operations), since U_{perf} is not a member at the start of the operation. The joining members also provide the joining private data produced by KC.PrepareJoin and a state unlocker (see Section 10.1.9) as parameters. All agents executing the operation, including the joining members, must begin with the same $PS \neq \perp$. The joining members begin with $SS = \perp$.

Parameters: T', a joining commit; Q^*, either joining private data or $\perp$; and U, either a state unlocker or $\perp$.
Output: U^*, a set of the agent IDs of the joining members; V^*, a mapping from new agent IDs to the invitation IDs used to join; I^*, a mapping from newly observed agent IDs to unblinded identity public keys; and A^*, the set of agent IDs in I^* that are authenticated.

Execute KC.MergeJoin with:
- Parameter T': set to T'
- Parameter Q^*: set to Q^*
- Output B^*: store in B^*.

$(R_1, B, R_2) \leftarrow$ MLS.DeniabilitySet(PS, $\perp$).
for each $((aid, iid, bpk, bsk, \overline{pk}, \pi) \in B^*)$ {
  Add aid to U^*.
  Add $aid \rightarrow iid$ to V^*.
  if (the session is not in "non-repudiable" mode) { Append personal(aid) to R_1. }
  if (the session is in "strongly deniable" mode) {
    Append $\underline{bpk}$ to B.
    Append $\overline{pk}$ to R_2.
  }
}
for each $((aid, iid, bpk, bsk, \overline{pk}, \pi) \in B^*$, sorted by $aid)$ {
  Locate the layaway $(iid, ipk, e, btpk, \overline{tpk}, u)$ with identifier iid in $PS.L$.
  if (if no such layaway exists) { Abort. }
  if ($e \neq \perp$ **&&** e has expired) { Abort. }

$m \leftarrow$ personal(aid).
if (Auth.Verify($ipk, bpk, \overline{pk}, btpk, \overline{tpk}, R_1, B, R_2, m, \pi$) = 0) { Abort. }
if ($u \neq \perp$) { Update the layaway with identifier iid in $PS.L$ with $u \leftarrow u - 1$. }
if (($u \neq \perp$ **&&** $u < 1$) || $btpk \neq \perp$) {
    Remove the layaway with identifier iid from $PS.L$.
}
}
if (U_{proc} is one of the joining members) {
Initialize SS with the correct structure (see Section 10.1.2).
if (U contains an invitation ID iid and invitation private key isk) {
    $SS \leftarrow$ LVSM.E.Decrypt(PS, SS, iid, isk).
    if ($SS = \perp$) { Abort. }
    $SS \leftarrow$ LVSM.U.Decrypt(PS, SS, iid, isk).
    if ($SS = \perp$) { Abort. }
} **else if** (U contains label-value secrets E and unblinding secrets U) {
    $SS.E \leftarrow E$.
    $SS.U \leftarrow U$.
} **else** {
    Abort.
}
for each ($aid \rightarrow (\overline{pk}, bpk)$ in $PS.I$) {
    if ($aid \in U^*$) { **continue** }
    $pk \leftarrow$ Unblind($\overline{pk}, bpk, SS.U[bpk]$).
    Add $aid \rightarrow pk$ to I^*.
}
}
Execute LVSM.E.RefreshInvites.
if ($PS.U.f = 0$) {
Execute LVSM.U.RefreshEncryption.
} **else** {
Execute LVSM.U.RefreshInvites.
}
if (long-term identities are enabled) {
for each ($(aid, iid, bpk, bsk, \overline{pk}, \pi) \in B^*$, sorted by aid) {
    if (bpk does not appear in $PS.I$) {
        Execute LVSM.U.Set with:
        ▸ Parameter bpk: set to bpk

▸ Parameter *bsk*: set to *bsk*.
}
$PS.I[aid] \leftarrow (\overline{pk}, bpk)$.
if $(bsk \neq \perp)$ {
$pk \leftarrow \text{Unblind}(\overline{pk}, bpk, bsk)$.
Add $aid \rightarrow pk$ to I^*.
Add aid to A^*.
}
}
}
Execute MLS.CompleteOperation with:
▸ Parameter D: set to ("MassJoin", T')
▸ Parameter b: set to 0.

10.10.5 Mass Evict

The **MLS.MassEvict** operation allows a member to evict a set of other members from the session, preventing them from decrypting future payloads. It is not possible for a member to evict themselves; see Section 10.12 for a discussion of this limitation. If U_{proc} is one of the evicted members, then $SS = \perp$ once the operation has been received.

Parameters: U, a set of agent IDs specifying members to evict.

Execute KGM.Evict with:
▸ Parameter U: set to U
▸ Parameter *bctx*: set to *bctx*
▸ Output *bctx*: store in *bctx*.
for each ($u \in U$, sorted) {
Execute KGM.DiscardMember with:
▸ Parameter U: set to u.
Delete any mapping from u in $PS.I$.
}
for each (*bpk* that previously appeared but no longer appears in $PS.I$) {
Execute LVSM.U.Delete with:
▸ Parameter *bpk*: set to *bpk*.
}
Execute LVSM.E.MarkStale.

Execute LVSM.U.MarkStale.
Execute MLS.CompleteOperation with:
- Parameter D: set to ("MassEvict", U)
- Parameter b: set to 1.

10.10.6 fs-Ratchet

The **MLS.FSRatchet** operation is a lightweight operation that derives a new group key.

$PS.H \leftarrow H(PS.H \| U_{\text{perf}} \| \text{SerializePS}(PS))$
$\hookrightarrow$ where H is a domain-separated cryptographic hash function.
Execute KGM.RatchetGroupKey.

10.10.7 pcs-Ratchet

The **MLS.PCSRatchet** operation allows a member to replace all of their ephemeral keys with new ones. This is used to achieve post-compromise security by replacing any keys that an adversary may have previously compromised. The label-value stores are marked as stale because in order to provide post-compromise security, it must be assumed that an adversary may have compromised $SS.E$ or $SS.U$.

Execute CKeyGen with:
- Output pk^*: store in pk^*
- Output sk^*: store in sk^*.

Update personal(U_{perf}) $\rightarrow pk^*$ in $PS.P$.
if ($U_{\text{proc}} = U_{\text{perf}}$) {
  Update personal(U_{perf}) $\rightarrow sk^*$ in $SS.Q$.
}
$R \leftarrow \text{keyset}_{PS.G}(U_{\text{perf}}) \setminus \{\text{personal}(U_{\text{perf}})\}$.
Execute MLS.ReplaceKeys with:
- Parameter R: set to R.

Execute LVSM.E.MarkStale.
Execute LVSM.U.MarkStale.
Execute MLS.CompleteOperation with:
- Parameter D: set to ("PCSRatchet")
- Parameter b: set to 1.

10.10.8 Cleanup

The **MLS.Cleanup** operation erases superfluous edges to any k-nodes known to U_{perf}.

$R \leftarrow \emptyset$.
for each ($r \in \text{keyset}_{PS.G}(U_{\text{perf}})$) {
  if (there exists a mapping to r in $PS.S$) {
    $R \leftarrow R \cup \{r\}$.
  }
}
Execute MLS.ReplaceKeys with:
- Parameter R: set to R.

Execute MLS.CompleteOperation with:
- Parameter D: set to ("Cleanup")
- Parameter b: set to 1.

10.10.9 pcs-Reidentify

The **MLS.PCSReidentify** operation allows a member to replace their long-term identity with a new one. The purpose of this operation is to provide post-compromise security in the case where the member's identity private key has been compromised. Trust establishment will need to be performed again for the new identity public key. U_{perf} provides the new unblinded identity public key and the new identity witness as parameters to the operation, while receiving agents set these parameters to $\perp$.

Parameters: pk, either a new unblinded identity public key or $\perp$; and w, either an identity witness for pk or $\perp$.

---

if (long-term identities are not enabled) { Abort. }
$(R_1, B, R_2) \leftarrow$ MLS.DeniabilitySet($PS, \perp$).
$m \leftarrow$ SerializePS(PS).
Let bpk denote the most recently added blinding public key in $PS.I$.
if (performer context) {
  $bsk \leftarrow SS.U[bpk]$.
  $\overline{pk} \leftarrow$ Blind(pk, bpk, bsk).
  $\pi \leftarrow$ Auth.Prove($\perp, bpk, \overline{pk}, \perp, \perp, R_1, B, R_2, m, w$).
  Write $\overline{pk}$ and π to the commit.

} **else** {
  Read $\overline{pk}$ and π from the commit.
  if (Auth.Verify($\perp, bpk, \overline{pk}, \perp, \perp, R_1, B, R_2, m, \pi$) = 0) { Abort. }
}
Update $U_{\text{perf}} \rightarrow (\overline{pk}, bpk)$ in $PS.I$.
Execute MLS.CompleteOperation with:
- Parameter D: set to ("PCSReidentify", π)
- Parameter b: set to 1.

10.10.10 Authenticate

The **MLS.Authenticate** operation allows U_{perf} to prove its identity to members that recently joined the session. An overview of this mechanism was given in Section 10.1.9. U_{perf} provides its unblinded identity public key and an identity witness as parameters to the operation, while receiving agents set these parameters to $\perp$.

Parameters: U, the set of members that U_{perf} is authenticating to; pk, either the existing unblinded identity public key or $\perp$; and w, either an identity witness for pk or $\perp$.

---

if (long-term identities are not enabled) { Abort. }
$(R_1, B, R_2) \leftarrow$ MLS.DeniabilitySet(PS, U).
$m \leftarrow$ SerializePS(PS).
Locate $U_{\text{perf}} \rightarrow (\overline{pk}, bpk)$ in $PS.I$.
Let bpk' denote the most recently added blinding public key in $PS.I$.
if ($bpk = bpk'$) {
  $\underline{bpk}' \leftarrow \perp$.
  $\overline{pk}' \leftarrow \perp$.
} **else** {
  if (performer context) {
    $\underline{bsk}' \leftarrow SS.U[bpk']$.
    $\overline{pk}' \leftarrow \underline{\text{Blind}}(pk, bpk', bsk')$.
    Write $\overline{pk}'$ to the commit.
  } **else** {
    Read $\overline{pk}'$ from the commit.
  }
}

if (performer context) {
  $bsk \leftarrow SS.U[bpk]$.
  $\pi \leftarrow$ Auth.Prove$(\perp, bpk, \overline{pk}, bpk', \overline{pk}', R_1, B, R_2, m, w)$.
  Write π to the commit.
} **else** {
  Read π from the commit.
  if (Auth.Verify$(\perp, bpk, \overline{pk}, bpk', \overline{pk}', R_1, B, R_2, m, \pi) = 0$) { Abort. }
}
if $(bpk' \neq \perp)$ {
  Update $U_{\text{perf}} \rightarrow (\overline{pk}', bpk')$ in $PS.I$.
  for each (bpk that previously appeared but no longer appears in $PS.I$) {
    Execute LVSM.U.Delete with:
      ▸ Parameter bpk: set to bpk.
  }
}
Execute MLS.CompleteOperation with:
  ▸ Parameter D: set to ("Authenticate", U, π)
  ▸ Parameter b: set to 1.

10.10.11 Change Mode

The **MLS.ChangeMode** operation replaces the developer-supplied mode $PS.M$ with a new one. The group policy may alter the configurable behavior described in Section 10.1.6 based on the new mode string. Only certain configuration changes are possible: sessions cannot become more deniable, and long-term identities cannot be introduced after running in ephemeral mode.

Parameters: M, the new mode string.

(Deniability, LongTermIDs, InviteAnonymity, PCS) $\leftarrow$ SessionConfig($PS.M$).
(Deniability′, LongTermIDs′, InviteAnonymity′, PCS′) $\leftarrow$ SessionConfig(M).
if (**!** LongTermIDs′ **&&** Deniability′ $\neq$ "strongly deniable") { Abort. }
if (Deniability′ is a more deniable mode than Deniability) { Abort. }
if (**!** LongTermIDs **&&** LongTermIDs′) { Abort. }
$PS.M \leftarrow M$.
if (LongTermIDs **&& !** LongTermIDs′) {
  Set $PS.I$ to an empty map.

```
    for each (bpk such that PS.U[bpk] exists) {
        Execute LVSM.U.Delete with:
          ▸ Parameter bpk: set to bpk.
    }
}
Execute MLS.CompleteOperation with:
  ▸ Parameter D: set to ("ChangeMode", M)
  ▸ Parameter b: set to 1.
```

10.10.12 Change Developer Data

Several operations are made available to the developer to modify the contents of the public label-value store, $PS.A$, and the confidential label-value store, $PS.E$. As with all operations, the developer is responsible for providing the parameters for the operations that modify the label-value stores (see Section 10.1.1). In particular, the developer must arrange for all of the members to learn the label that is being targeted by the operation. The label could be transmitted explicitly as part of the higher-level protocol, or it could be inferred from some other external event.

10.10.12.1 Set Public Data

The **MLS.SetPublicData** operation modifies a value stored in $PS.A$. The purpose of the operation is for the performer to store a new value in the group state; the other members learn this value when applying the commit. Consequently, the performing member provides the new value as a parameter, while receiving agents leave the parameter empty.

Parameters: ℓ, the label; and v, either the new value or $\perp$.

```
if (performer context) {
    Write v to the commit.
} else {
    Read v from the commit.
}
PS.A[ℓ] ← v.
Execute MLS.CompleteOperation with:
  ▸ Parameter D: set to ("SetPublicData", ℓ)
  ▸ Parameter b: set to 1.
```

10.10.12.2 Delete Public Data

The **MLS.DeletePublicData** operation removes a value from $PS.A$.

Parameters: ℓ, the label to remove.

Delete $PS.A[\ell]$, if it exists.
Execute MLS.CompleteOperation with:
- Parameter D: set to ("DeletePublicData", ℓ)
- Parameter b: set to 1.

10.10.12.3 Set Confidential Data

The **MLS.SetConfidentialData** operation modifies a value stored in $PS.E$. The performing member provides the new value as a parameter, while receiving agents leave the parameter empty.

Parameters: ℓ, the label; and v, either the (unencrypted) new value or $\perp$.

Execute LVSM.E.Set with:
- Parameter ℓ: set to ℓ
- Parameter v: set to v.

Execute MLS.CompleteOperation with:
- Parameter D: set to ("SetConfidentialData", ℓ)
- Parameter b: set to 1.

10.10.12.4 Delete Confidential Data

The **MLS.DeleteConfidentialData** operation removes a value from $PS.E$.

Parameters: ℓ, the label to remove.

Execute LVSM.E.Delete with:
- Parameter ℓ: set to ℓ.

Execute MLS.CompleteOperation with:
- Parameter D: set to ("DeleteConfidentialData", ℓ)
- Parameter b: set to 1.

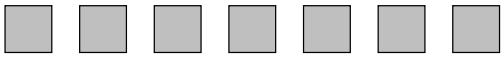
10.11 Key Control Schemes

The KC subsystem is an important part of Safehouse that is primarily responsible for determining the shape of the key graph, *PS.G*. Section 10.6 defined the functions, operations, and protocols that must be provided by the KC subsystem. The KC subsystem is separated from the other parts of Safehouse because although many radically different schemes are possible, selecting a scheme is entirely a matter of performance optimization—not cryptography or security. This section provides high-level descriptions of some intuitive and efficient schemes in order to demonstrate that it is practical to use Safehouse in the target scenarios (described in Chapter 7). Improving upon these initial schemes is certainly possible, but it is fundamentally an algorithmic optimization problem that is outside of the scope of this work.

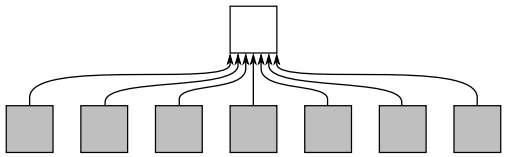
The following four KC schemes are described in this section:

- **Trivial** (Section 10.11.1): the key graph contains no *k*-nodes except for the members' personal keys. This is the most basic possible KC scheme that uses BRAKEM. The Trivial scheme provides a lower bound on how difficult it is to implement the KC subsystem.
- **Star** (Section 10.11.2): the key graph consists of the members' personal keys and one additional *k*-node shared by all members. The shared *k*-node makes MLS.CompleteOperation (see Section 10.10.1.1) more efficient than in the Trivial scheme: KGM.RefreshKeys (see Section 10.5.8), which is executed as a subroutine, can refresh exhausted keys with a BRAKEM transfer to a single recipient, instead of *n* recipients for a session with *n* members.
- **Shrub** (Section 10.11.3): the *u*-nodes in the key graph are partitioned into sets with a given maximum size, each sharing an intermediate *k*-node, with a root *k*-node shared by all sets. The motivation for this scheme compared to the Star scheme is that it allows the root key to be replaced far more efficiently as the group size increases: the BRAKEM transfer is usually able to use the intermediate *k*-nodes as recipients instead of the personal keys, reducing the cost of the BRAKEM by a constant factor.
- **Tree** (Section 10.11.4): a key graph is arranged in a generic tree structure. This scheme is parameterized by a function that defines the maximum branching factor at each depth. It uses heuristics to try to keep the tree as shallow as possible over time.

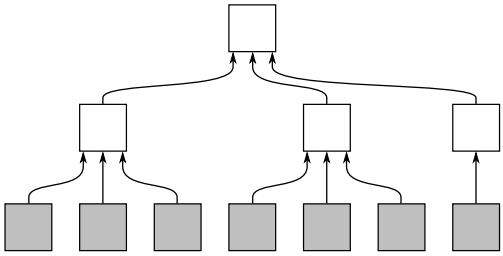
Figure 10.10 depicts some possible key graphs produced by the schemes in order to illustrate the difference between them at a glance. Aside from the Trivial scheme, all of the KC schemes in this section use TKLL (see Chapter 9) to establish shared keys during the KC.PrepareJoin protocol. The following sections describe the schemes in terms of the KC interface (see Section 10.6).



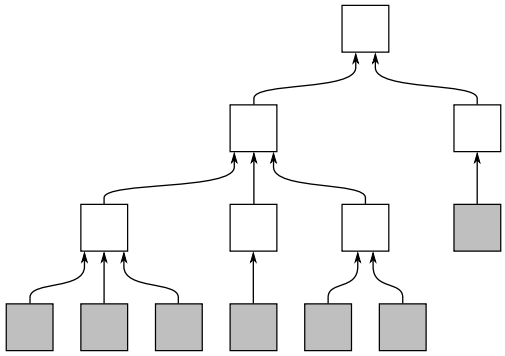
(a) THE TRIVIAL SCHEME. The graph consists of only personal keys.



(b) THE STAR SCHEME. An extra *k*-node is shared by the personal keys.



(c) THE SHRUB SCHEME. This example uses a maximum set size of three. Each set of personal keys shares an intermediate *k*-node, and one root *k*-node is shared by all intermediate *k*-nodes.



(d) THE TREE SCHEME. This scheme can produce arbitrary tree structures. This example uses a constant maximum branching factor of three.

Figure 10.10 EXAMPLES OF KEY GRAPHS PRODUCED BY THE KEY CONTROL SCHEMES. Shaded squares represent personal keys, each with an implicitly attached *u*-node. Unshaded squares represent *k*-nodes that are not personal keys. All key graphs are for groups with seven members. These are not necessarily the simplest graphs produced by the schemes; they are examples of possible states. (Ref: 428)

10.11.1 Trivial Key Control

The Trivial scheme creates key graphs in which the only k-nodes are personal keys. It does not maintain any structural information in its state.

10.11.1.1 Initialization

To initialize a session, the Trivial scheme simply creates a key graph with a single personal key. The initialization message contains only the public key for the k-node. The initialization functions operate as follows:

- **KC.InitGraphPerform**$(U) \rightarrow (G, P, Q, T)$: Use CBRAKEM.KeyGen to generate a new key pair (ppk, psk). Create a key graph, G, with one u-node bound to agent ID U and an attached personal key bound to public key ppk. Store a mapping from the k-node to psk in Q. Set $T \leftarrow pk$.
- **KC.InitGraphApply**$(U, T) \rightarrow (G, P)$: Create a key graph with one u-node bound to U and a personal key bound to public key $pk = T$.

10.11.1.2 k-node Merging

Since the Trivial scheme does not create any intermediate k-nodes, KGM.Evict (see Section 10.5.7) never deletes any keys. Consequently, **KC.MayMerge** is never called and thus no implementation is necessary.

10.11.1.3 Serialization

The serialization functions **KC.Serialize** and **KC.Deserialize** are easy to optimize. KC.Serialize sorts the u-nodes by increasing agent ID. For each u-node in sorted order, KC.Serialize outputs the agent ID, the personal public key, and the blinding public key and blinded public key of the associated member (if long-term identities are enabled). It is not possible for superfluous edges to exist in key graphs produced by the Trivial scheme, so KC.Deserialize always returns $S = \emptyset$.

10.11.1.4 Shared Key

Since there are no shared k-nodes, **KC.BroadcastKeys** returns a set of all of the personal keys. For the same reason, **KC.SharedKey** returns $\perp$ to indicate that there is no k-node shared by all members.

10.11.1.5 Joining Procedure

The **KC.PrepareJoin** construction for the Trivial scheme is the only one in this chapter that does not make use of TKLL. The protocol proceeds as follows:

1. If long-term identities are enabled for the session, then each joining member:
   a) generates a new blinding key pair (bpk, bsk) using CBRAKEM.KeyGen;
   b) blinds its long-term identity pk to produce $\overline{pk}$;
   c) encapsulates the blinding private key bsk to the existing members and other joining members using SBRAKEM.Encapsulate (see Section 10.3); and
   d) sends the blinding public key bpk, blinded public key $\overline{pk}$, and the SBRAKEM encapsulation to the other joining members.
2. Each joining member uses Auth.Prove (see Section 10.8.3) with its identity witness w to produce an appropriate authentication NIZKPK based on the session configuration and the other joining members' blinded public keys, as described below.
3. The members exchange authentication NIZKPKs.

The joining commit consists of the following fields, grouped by joining member:

- the agent ID, aid;
- the invitation to use, identified by iid;
- the personal public key ppk;
- the blinding public key, bpk;
- the SBRAKEM encapsulation of the blinding private key;
- the blinded public key, $\overline{pk}$; and
- the authentication NIZKPK, π.

The identity blinding steps are skipped if long-term identification is not enabled for the session. Each member produces its authentication NIZKPK π as follows:

Retrieve $(iid, ipk, e, btpk, \overline{tpk}, u)$ from $PS.L$ for the given iid.
$(R_1, B, R_2) \leftarrow$ MLS.DeniabilitySet$(PS, \perp)$.
for each (joining member identified by $aid' \neq aid$) {
  if (the session is not in "non-repudiable" mode) { Append personal(aid') to R_1. }
  if (long-term identities are enabled **&&** the session is in "strongly deniable" mode) {
    Let $\underline{bpk}'$ be the blinding public key for the member.
    Let $\overline{pk}'$ be the blinded public key for the member.
    Append $\underline{bpk}'$ to B.
    Append $\overline{pk}'$ to R_2.
  }
}
$m \leftarrow$ personal(aid).
$\pi \leftarrow$ Auth.Prove$(ipk, bpk, \overline{pk}, btpk, \overline{tpk}, R_1, B, R_2, m, w)$.

The joining private data contains only the personal private key.

The construction of **KC.MergeJoin** is straightforward. The joining commit is processed to extract the aforementioned data. For each member:

1. A u-node is added to $PS.G$ with agent ID aid.
2. A personal key for the u-node is added to $PS.G$ and bound to ppk in $PS.P$.
3. If U_{proc} is an existing member or joining member, it decapsulates the blinding private key bsk using SBRAKEM.Decapsulate. Otherwise, it sets bsk to $\perp$ and verifies the encapsulation using SBRAKEM.Verify.
4. $(aid, iid, bpk, bsk, \overline{pk}, \pi)$ is added to B^*.

KC.MergeJoin then returns the new member data in B^*.

10.11.2 Star Key Control

The Star scheme is like the Trivial scheme, except that it adds a single shared k-node to the key graph. This k-node is called the *root key*. An implementation of the Star scheme might keep a reference to the root key in its state for convenience, but this is a small optimization: it is always possible to find the root key in constant time given a reference to any k-node in the graph by simply following one outgoing edge, if one exists.

10.11.2.1 Initialization

Initialization is identical to the Trivial scheme described in Section 10.11.1, except that an extra key pair (rpk, rsk) for the root key is generated with CBRAKEM.KeyGen. A k-node is added to the key graph with an edge from the personal key to this new root key. The root key is mapped to rpk in the public key mapping. rpk is included in the initialization message T. Within **KC.InitGraphApply**, rpk is extracted from T. A root key is added to the key graph in the same way and mapped to rpk in the public key mapping.

10.11.2.2 k-node Merging

Since the root key is shared by all personal keys, the only situation in which **KC.MayMerge** would be called is if an KGM.Evict operation evicts all members except for U_{perf}. There is no good reason to discard the root key in this scenario, and doing so would make the scheme more complicated. Consequently, KC.MayMerge always returns $b = 0$ for the Star scheme.

10.11.2.3 Serialization

Serialization for the Star scheme is identical to the Trivial scheme, except that the public key for the root key, rpk, is appended to the serialization by **KC.Serialize**. In **KC.Deserialize**, rpk is extracted from the data. Next, KC.Deserialize adds a root key to the recovered key graph, and adds edges from all personal keys to the root key. It then maps the root key to rpk in the public key mapping.

10.11.2.4 Shared Key

The main advantage of the Star scheme over the Trivial scheme is the presence of a shared k-node in order to make key updates far more efficient. When **KC.BroadcastKeys** is called, the Star scheme returns a set containing only the root key. This dramatically improves the performance of MLS.CompleteOperation (see Section 10.10.1.1), LVSM.E.RefreshEncryption (see Section 10.7.1.1.3), and LVSM.U.RefreshEncryption (see Section 10.7.1.3.3). Similarly, **KC.SharedKey** returns the root key for the Star scheme.

10.11.2.5 Joining Procedure

The Star scheme uses authenticated TKLL (see Section 9.4) to implement the **KC.PrepareJoin** and **KC.MergeJoin** process. When a mass join operation begins, the existing key graph for the

session always contains only a root key, *rpk*, and personal keys for the existing members. The KC.PrepareJoin protocol proceeds as follows:

1. Construct a new key tree, T^*, that contains personal keys for the joining members connected to a *merging root key*, rpk^*. If long-term identities are enabled for the session, add another k-node called bpk^* to the tree, and add an edge from rpk^* to bpk^*.
2. The joining members use authenticated TKLL to establish key pairs for T^*.

The authenticated TKLL protocol used by KC.PrepareJoin must be given access to four functions: Sig, SVerif, Compute-ID, and Prove-ID. The definition of these functions is given in Section 9.4.1. The Star scheme instantiates TKLL using the following implementations of the required functions:

- **Sig**$(pk, sk, m) \rightarrow \sigma$: this function is implemented using the Sig function for the chosen BRAKEM key space, as described in Section 10.9.3.
- **SVerif**$(pk, m, \sigma) \rightarrow \{0, 1\}$: this function is implemented using the SVerif function for the chosen BRAKEM key space, as described in Section 10.9.3.
- **Compute-ID**$(P^*, Q^*) \rightarrow I$: this function generates a long-term identity for the calling member. If long-term identities are disabled for the session, then the function returns $I = \perp$. Otherwise, the function locates the blinding public key bpk^* in P^* and the corresponding blinding private key bsk^* in Q^*. It uses these values to compute the blinded public key $I \leftarrow \text{Blind}(pk, bpk^*, bsk^*)$ using the Blind function described in Section 10.1.8 with the member's identity public key, *pk*.
- **Prove-ID**$(I^*, P^*, Q^*) \rightarrow \pi$: this function produces an authentication NIZKPK for the member. The process is similar to the Trivial scheme, except that some values are extracted from the function's parameters instead of the public state. The implementation produces the authentication NIZKPK using the calling member's agent ID, *aid*, its (unblinded) identity public key, *pk*, and its identity witness, *w*. The function proceeds as follows:

> Locate the blinding public key bpk^* in P^*.
> Locate the blinding private key bsk^* in Q^*.
> Retrieve $(iid, ipk, e, btpk, \overline{tpk}, u)$ from $PS.L$ for the caller's *iid*.
> $(R_1, B, R_2) \leftarrow \text{MLS.DeniabilitySet}(PS, \perp)$.
> **for each** (mapping $aid' \rightarrow \overline{pk'}$ in I^*) {
> **if** $(aid' = aid)$ { **continue** }

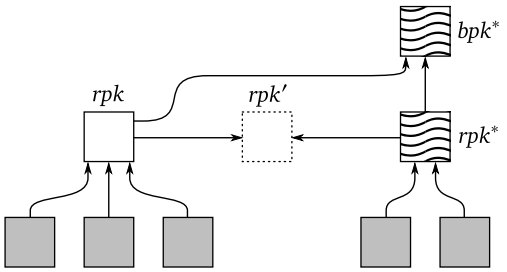
```
    if (the session is not in "non-repudiable" mode) { Append personal(aid′) to R_1. }
    if (long-term identities are enabled && session is in "strongly deniable" mode) {
        Append bpk* to B.
        Append pk̄′ to R_2.
    }
}
m ← personal(aid).
Locate the mapping aid → pk̄ in I*.
π ← Auth.Prove(ipk, bpk*, pk̄, btpk, t̄pk, R_1, B, R_2, m, w).
return π.
```

The output of the TKLL protocol includes a mapping from the joining members' agent IDs to blinded public keys in I^* and to authentication NIZKPKs in Π^*. These mappings are used to produce the joining commit. Additionally, the output of TKLL includes a multi-signature, σ, that demonstrates that all joining members accept the outcome of the protocol. The joining commit consists of:

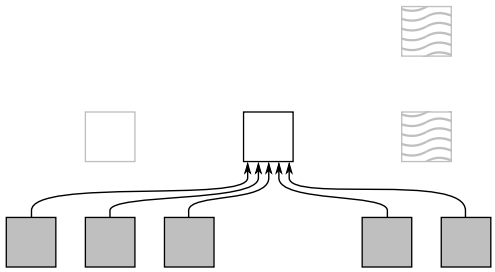
- For each joining member:
  - the agent ID, *aid*;
  - the invitation to use, identified by *iid*;
  - the personal public key *ppk*;
  - the blinded public key, $\overline{pk}$, extracted from I^*; and
  - the authentication NIZKPK, π, extracted from Π^*.
- The multi-signature for the TKLL output, σ.
- The public key for each non-personal k-node in T^*, traversed in a deterministic order.

The joining private data contains Q^*, the mapping from k-nodes to private keys produced by TKLL. The KC.MergeJoin operation proceeds as follows:

1. Verify that the multi-signature σ in the joining commit is valid using the MSBN.Vf function described in Section 9.4.1.
2. Compute the shape of T^* as in KC.PrepareJoin.



(a) BEFORE KGM.DISCARDKEY. The merging root key, rpk^*, is merged with the existing root key, rpk, to form the new root key, rpk'. bpk^* is used as the blinding public key for the joining members. An edge is added from rpk to bpk^* using KGM.ShareKey.



(b) AFTER KGM.DISCARDKEY. The k-nodes for bpk^*, rpk, and rpk^* are discarded, creating a new key graph with root key rpk' that satisfies the Star scheme's intended shape.

Figure 10.11 JOINING PROCEDURE WITH THE STAR KC SCHEME. Shaded squares represent personal keys. Normal squares represent existing k-nodes. Squares with wave patterns represent k-nodes established by TKLL. Squares with dashed borders represent new k-nodes created by KGM.MergeKeys during KC.MergeJoin. This example depicts two joining members being added to an existing session with three members. (Refs: 436 and 440[a,b])

3. The key tree instantiated by TKLL, T^*, is imported into the key graph using KGM.ImportTree (see Section 10.5.11). If the joining private data is not $\perp$, the private keys contained in Q^* are provided to KGM.ImportTree.

4. If long-term identities are enabled, then KGM.ShareKey (see Section 10.5.10) is executed to share bpk^* with rpk, which gives all existing members access to bsk^*. KGM.DiscardKey (see Section 10.5.6) is executed to discard bpk^* from the key graph.

5. KGM.MergeKeys (see Section 10.5.4) is executed to merge rpk and rpk^* into a *new root key*, rpk'.

6. KGM.DiscardKey is executed twice to discard rpk and rpk^*, causing all edges from the personal keys to connect to rpk'.

7. For each joining member, $(aid, iid, bpk^*, bsk^*, \overline{pk}, \pi)$ is added to B^*.

Figure 10.11 depicts the state of the key graph during KC.MergeJoin, both before and after k-nodes are discarded. The correctness of this approach follows from the careful design of the KGM-layer operations, and the use of TKLL to establish the new key tree ensures that the KC.PrepareJoin protocol is very efficient. Using TKLL to share bsk^* allows all of the joining

members to blind their identity public keys using the same blinding key pair. This reuse of a blinding key pair is another improvement compared to the Trivial scheme, in which each joining member uses its own blinding key pair, leading to more entries in $PS.U$. As existing members authenticate to the joining members over time, they will slowly adopt the newest blinding key pair (see Section 10.10.10).

10.11.3 Shrub Key Control

The Shrub scheme partitions the u-nodes in the key graph into sets. The scheme is parameterized by n_S, the *maximum set size*, which is an upper bound on the number of u-nodes in each set. For each set, the key graph contains a k-node called the *set key*. There are edges from each personal key in a set to the associated set key. Finally, there is a single k-node called the *root key*. There are edges from each set key to the root key, ensuring that all members know the private key for the root key. Like the Star scheme (see Section 10.11.2), it is easy to locate the root key and all of the set keys in constant time given a reference to any k-node, so it is not necessary for an implementation of the Shrub scheme to store stateful structural information. However, maintaining references to the root key and set keys may provide a marginal speed improvement in practice.

10.11.3.1 Initialization

The initialization procedure for the Shrub scheme creates a key graph with one personal key, one set key, and one root key. Consequently, **KC.InitGraphPerform** and **KC.InitGraphApply** are identical to the Trivial scheme (see Section 10.11.1), except that the performer generates two extra key pairs with CBRAKEM.KeyGen, adds both public keys to the initialization message, and adds the two additional k-nodes to the key graph. KC.InitGraphApply reads the two public keys and constructs the same key graph and public key mapping. This process is the same as for the Star scheme, except that two non-personal k-nodes are added instead of just one.

10.11.3.2 k-node Merging

The Shrub scheme never allows merging keys during evictions. The KGM.Evict operation will only request a k-node merger if all but one member has been evicted from a set. These mergers are disallowed because this would cause some personal keys to become direct children of the root key instead of a set key. This would require additional complexity in order to restore the set key when new members join the group, with very little performance benefit in return. Consequently, **KC.MayMerge** always returns $b = 0$ for the Shrub scheme.

10.11.3.3 Serialization

Serialization is slightly more complicated for the Shrub scheme than for the Trivial or Star schemes. **KC.Serialize** first encodes the public key for the root key. Next, KC.Serialize groups together sets by the number of u-nodes contained in them. For each cardinality n_i, with $1 \leq n_i \leq n_S$, KC.Serialize encodes the cardinality and an array of sets with that cardinality. For each set in this array, KC.Serialize encodes the public key for the stem key, an optional list of agent IDs that have superfluous knowledge of the set key (based on the contents of *PS*), and a list of u-nodes contained in the set. For each u-node in a set, KC.Serialize encodes the agent ID of the member, the personal public key, and the blinding public key and blinded public key (if long-term identities are enabled). The arrays and lists in the serialization are sorted in a canonical order; in practice, the sets can be sorted by their position in the key tree, and the agent IDs and u-nodes can be sorted based on the numerical ordering of the agent IDs. Grouping the sets together by cardinality in the serialization can result in a significant size reduction for large groups, since most sets will contain n_S members.

10.11.3.4 Shared Key

KC.BroadcastKeys for the Shrub scheme returns a set containing only the root key. **KC.SharedKey** returns the root key itself. This is the same as the Star scheme.

10.11.3.5 Joining Procedure

The joining procedure for the Shrub scheme is essentially the same as for the Star scheme, except that the key trees instantiated by TKLL are more complicated. The **KC.PrepareJoin** protocol behaves very similarly as in the Star scheme, except that it must take set keys into account. KC.PrepareJoin for the Shrub scheme proceeds as follows:

1. Prepare a mapping from joining members to the sets that will contain them. For each joining member, check to see if there is an existing set containing fewer than n_S members. If so, add a mapping from the joining member to this existing set. Otherwise, initialize a new set and add a mapping from the joining member to the new set.

2. Construct a new key tree, T^*, containing the personal keys for the joining members. Add a *merging set key* k-node, spk_i^*, for each set with index i in the joining member mapping. Add a *merging root key* k-node, rpk^*. Add edges from each personal key to its associated merging set key, spk_i^*, and from every spk_i^* to rpk^*. If long-term identities are enabled for the session, add another k-node called bpk^* to the tree, and add an edge from rpk^* to bpk^*.

3. The joining members use authenticated TKLL to establish key pairs for T^*, using the same definitions for Sig, SVerif, Compute-ID, and Prove-ID as for the Star scheme.

The joining commit is produced exactly as for the Star scheme: it contains the per-member data for each joining member, the multi-signature σ produced by TKLL, and the public key for each non-personal k-node in T^*, traversed in a deterministic order. The joining private data contains Q^*, the mapping from k-nodes to private keys produced by TKLL. The **KC.MergeJoin** operation differs from the Star scheme in that the merging set keys that correspond to existing set keys are combined using KGM.MergeKeys, while merging set keys for completely new sets are simply imported into the key graph. KC.MergeJoin proceeds as follows:

1. Verify that the multi-signature σ in the joining commit is valid using the MSBN.Vf function (see Section 9.4.1).

2. Compute the shape of T^* and the mapping from joining members to set keys using the same algorithm as KC.PrepareJoin.

3. The key tree instantiated by TKLL, T^*, is imported into the key graph using KGM.ImportTree (see Section 10.5.11). If the joining private data is not $\perp$, the private keys contained in Q^* are provided to KGM.ImportTree.

4. If long-term identities are enabled, then KGM.ShareKey (see Section 10.5.10) is executed to share bpk^* with rpk. KGM.DiscardKey (see Section 10.5.6) is executed to discard bpk^* from the key graph.

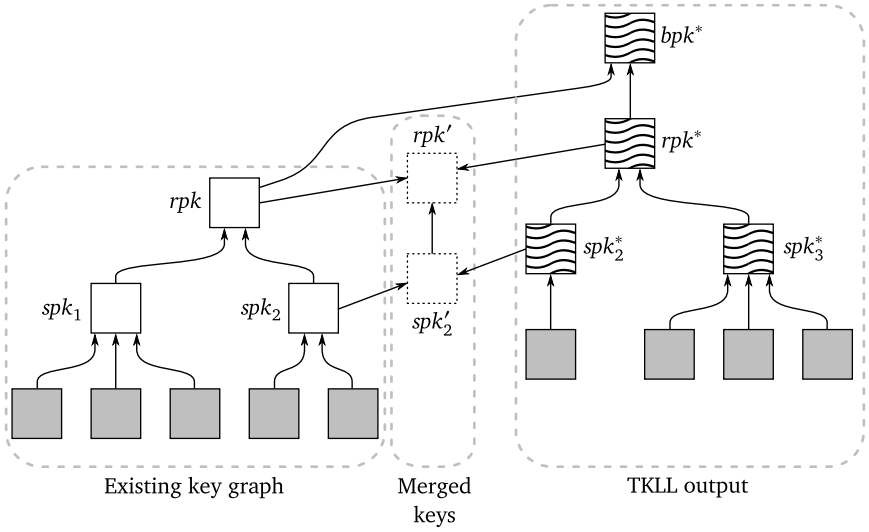
5. KGM.MergeKeys is executed to merge rpk and rpk^* into a *new root key*, rpk'.

6. For each existing set referenced in the joining member mapping, let spk_i denote the existing set key in $PS.G$ and spk_i^* denote the corresponding merging set key in T^*. KGM.MergeKeys (see Section 10.5.4) is executed to merge spk_i and spk_i^* into a *new set key*, spk_i'. For any new set keys created in this way, KGM.AddEdge (see Section 10.5.5) is executed to add an edge from spk_i' to rpk'.

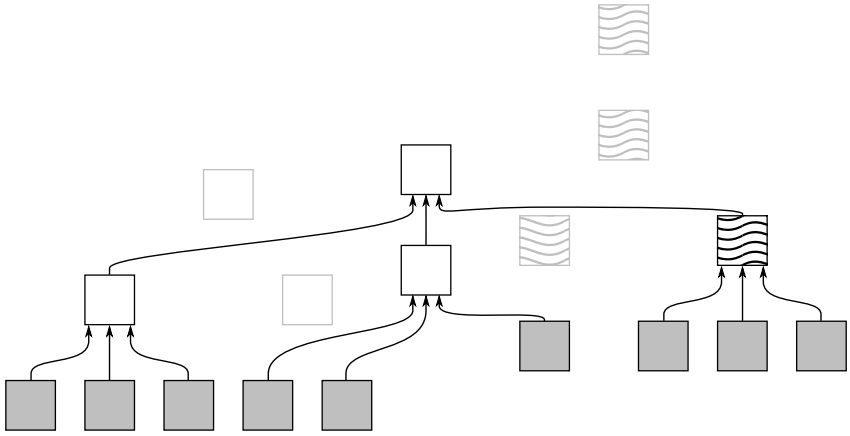
7. KGM.DiscardKey is executed twice to discard rpk and rpk^*, causing all edges from the personal keys to connect to rpk'.

8. For each merging set key spk_i^* that corresponds to an existing set key spk_i, KGM.DiscardKey is executed twice to discard spk_i and spk_i^*.

9. For each joining member, $(aid, iid, bpk^*, bsk^*, \overline{pk}, \pi)$ is added to B^*.



(a) BEFORE KGM.DISCARDKEY. rpk' is produced and bsk^* is shared as in the Star scheme, depicted in Figure 10.11. Additionally, the merging set key spk_2^* and set key spk_2 are merged to produce the new set key spk_2', essentially adding one joining member to the existing set. KGM.AddEdge connects spk_2' to rpk'.



(b) AFTER KGM.DISCARDKEY. This final key graph is produced when the k-nodes are discarded in the following order: bpk^*, rpk, rpk^*, spk_2, and then spk_2^*. The edge from spk_2' to rpk' that was previously added prevents edges from being added from spk_2 and spk_2^* to rpk' when rpk and rpk^* are discarded, which would be semantically equivalent but syntactically undesirable.

Figure 10.12 JOINING PROCEDURE WITH THE SHRUB KC SCHEME. The graph is depicted with the same graphical conventions as in Figure 10.11. This example depicts a session with five existing members, four joining members, and a maximum set size of $n_S = 3$. One new u-node is placed in the set with set key spk_2, while the other three new u-nodes are placed in a new set. After the KC.MergeJoin operation, the three set keys are $\{spk_1, spk_2', spk_3^*\}$ and the root key is rpk'. (Refs: 441 [a] [b])

Figure 10.12 depicts an example of the joining process both before and after keys are discarded using KGM.DiscardKey executions. The procedure is mainly the same as in the Star scheme, except that the edges for the set keys must be carefully managed. Using KGM.AddEdge to add edges from new set keys to the new root key is important to ensure that there are no edges from personal keys directly to rpk' at the end of the KC.MergeJoin operation. Adding an edge of this form is always permitted, because the two k-nodes with edges to a new set key are the (existing) set key and the merging set key, and these have outgoing edges to the (existing) root key and the merging root key, respectively; this means that all k-nodes with connections to a new set key already have alternate paths to the target of the sole outgoing edge, so the added edge is semantically redundant. The final graph depicted in Figure 10.12 is a tree with the correct form, with one joining member added to an existing set, and three joining members added to a new set. In total, the KC.MergeJoin operation only requires three CBRAKEM encapsulations for this example: one to share the blinding private key with the existing group, one to establish the new root key, and one to establish the new set key for the updated set.[32]

10.11.4 Tree Key Control

The Tree scheme arranges the key graph in a tree structure. It attempts to keep the tree as shallow as possible as members join and leave the session over time. The scheme is parameterized by a *branching factor function*, $bf(d)$, that determines the maximum number of children at each tree depth, d. The branching factor function must be monotonically increasing (i.e., the maximum number of children for a k-node at depth d is at least as many as for a k-node at depth $d-1$). The scheme also tries to preserve (but does not guarantee) a property: the children of every non-personal k-node are either all personal k-nodes or all non-personal k-nodes; this property is not strictly necessary, but it allows the joining procedure to use simpler techniques more often. Aside from these constraints, the graphs produced by the scheme are fully generalized key trees. Figure 10.13 depicts the key graph produced by the scheme for a session with 24 members with a branching factor function such that $bf(0) = 2$, $bf(1) = 3$, and $bf(2) = 4$. The scheme is designed to gracefully reduce the size of the tree if the number of members in the session decreases, so the performance of the scheme is not "poisoned" by leftover intermediate k-nodes after a large decrease in membership.

[32] The overall MLS.MassJoin operation (see Section 10.10.4) will require additional CBRAKEM encapsulations as layaway entries are claimed and exhausted keys are refreshed.

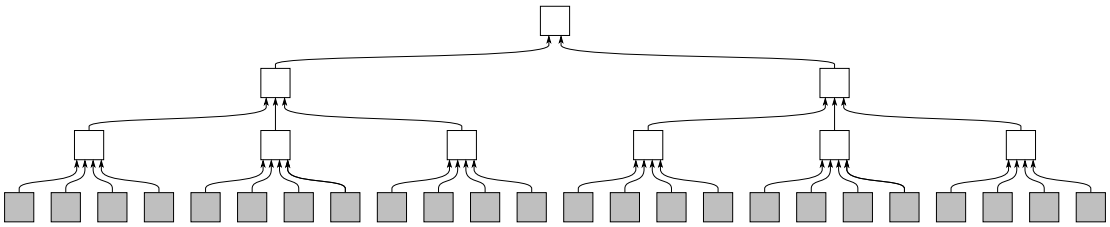


Figure 10.13 A COMPLETE KEY TREE WITH BRANCHING FACTORS $(2, 3, 4, \dots)$. With a height of four, this key tree contains 24 u-nodes. Shaded k-nodes are personal keys. (Ref: 441)

10.11.4.1 Initialization

The initialization procedure for the Tree scheme is the same as for the Trivial scheme, described in Section 10.11.1. Specifically, **KC.InitGraphPerform** and **KC.InitGraphApply** initialize a key graph containing only one u-node and one personal key. Unlike the Star and Shrub schemes, which both maintain a specific arrangement of intermediate k-nodes, the Tree scheme is capable of dynamically increasing the height of the key tree without limit. Consequently, it does not need to add any additional k-nodes to the initial group state.

10.11.4.2 k-node Merging

The **KC.MayMerge**(G, s, k) function for the Tree scheme allows k-nodes to be merged whenever possible, as long as the resulting tree does not violate the constraints imposed by the branching factor function. KC.MayMerge requires that an edge from s to k exists in G. In the context of the Tree scheme, this means that k is the parent of s. If s is initially at depth d, implying that k is at depth $d - 1$, then the subtree rooted at s must satisfy the branching factor function if it was positioned at depth $d - 1$. If and only if the branching factor function would be satisfied, KC.MayMerge returns $b = 1$. In general, determining if the branching factor function is satisfied requires traversing the entire subtree to check each k-node. If the branching factor is known to be constant at depth $d - 1$ and greater, then the condition is trivially satisfied.

10.11.4.3 Serialization

The serialization functionality for the Tree scheme must be capable of encoding arbitrary key trees. **KC.Serialize** encodes an array of blinding public keys used by the members (in a canonical

order[33]), followed by the key tree. The tree is encoded recursively, starting at the root node. For each node, a bit indicates if it is a *k*-node or a *u*-node. For *u*-nodes, the type bit is followed by the agent ID and, if long-term identities are enabled, the blinded public key and the index of the blinding public key in the initial array. If there are superfluous edges for the *u*-node's member, then the *u*-node data is followed by an array of superfluously known *k*-nodes. For *k*-nodes, the type bit is followed by the public key for the node and the recursively encoded child nodes. **KC.Deserialize** recursively decodes this data.

10.11.4.4 Shared Key

KC.BroadcastKeys for the Tree scheme returns a set containing only the *k*-node at the root of the tree. **KC.SharedKey** returns the root *k*-node itself. This is the same as the Star and Shrub schemes.

10.11.4.5 Joining Procedure

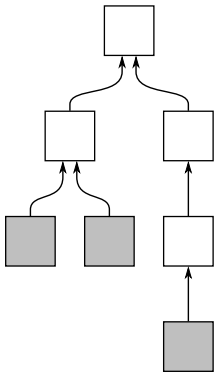
The high-level joining procedure for the Tree scheme is similar to the Star and Shrub schemes:

1. a new key tree is formed for the joining members;
2. the key tree is instantiated using authenticated TKLL within **KC.PrepareJoin**;
3. the instantiated key tree is imported into the existing key graph by KGM.ImportTree within **KC.MergeJoin**;
4. the old and new key trees are combined using KGM.MergeKeys and KGM.AddEdge; and then
5. certain *k*-nodes are discarded using KGM.DiscardKey to achieve the intended key graph shape.

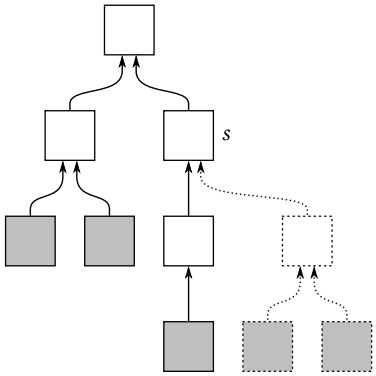
The main difference between the Tree scheme and the other schemes is that the key trees in the Tree scheme are generalized and may increase in height without any imposed bounds or structure (other than the branching factor function). Consequently, it is more complicated to form the new key tree and to merge the old and new trees.

Both KC.PrepareJoin and KC.MergeJoin begin by examining the existing key tree in *PS*.*G* and deciding where to add the newly joining members using the same deterministic algorithm. This

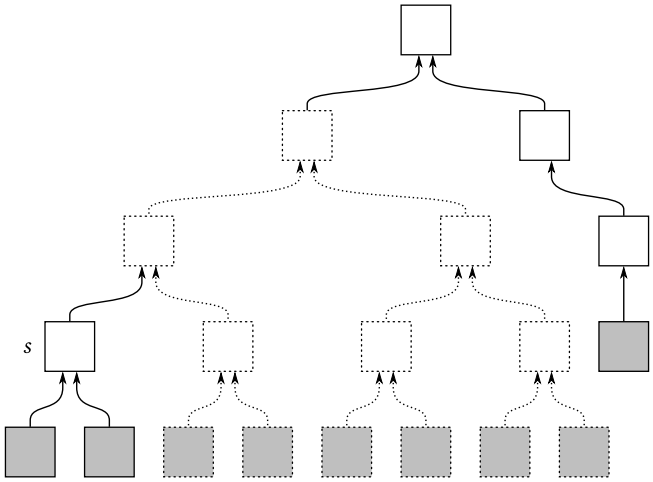
[33] In the prototype implementation, the blinding public keys are stored in *PS*.*I* in a sequence based on the order that they were introduced to the session. This order is included in the serialization. Other implementations can use alternative approaches, such as sorting the blinding public keys lexicographically during serialization.



(a) EXISTING KEY GRAPH.



(b) SUBTREE INSERTION. This option shows a subtree containing two *u*-nodes being inserted under the target *k*-node *s*.



(c) ROTATION. This option shows two new *k*-nodes being added on the edge from the root to the target *k*-node, *s*, with new subtrees contained under the new *k*-nodes.

Figure 10.14 NODE INSERTION OPTIONS FOR THE TREE KC SCHEME. Shaded *k*-nodes are personal keys. *k*-nodes and edges depicted with dashed lines indicate the newly proposed additions to the key tree. This example uses a constant maximum branching factor of two. (Ref: 445)

algorithm first produces a *node insertion plan* that describes a sequence of steps. Each step uses a parameterized *node insertion option* that modifies $PS.G$ in a specific manner to add new k-nodes and u-nodes. The Tree scheme uses two types of node insertion options: *subtree insertion*, and *rotation*. Examples of these options are depicted in Figure 10.14.

The subtree insertion option adds a new subtree as the child of an existing k-node. It accepts two parameters: the existing k-node that will become the new parent, s, and the number of u-nodes that should be contained in the subtree. s is called the *target* of the subtree insertion. This option is only applicable if $PS.G$ contains a k-node with fewer children than the maximum branching factor for its depth, which is not always the case. When employed, the subtree insertion option creates a complete subtree (i.e., all personal keys are located at the same depth and as far left as possible) with the given number of u-nodes, respecting the branching factor function for the intended depth. This subtree is then added as a child of the given parent k-node.

The rotation option adds one or more new nodes above a *target* node, s. The new nodes form a subtree, and s becomes a child in the leftmost path of this subtree. If s was the root, then the root of the newly added nodes becomes the new root for the entire key tree. If s was not the root, then the original parent k-node of s becomes the parent of the newly added nodes. The rotation option accepts two parameters: the target k-node s, and the number of u-nodes that should be contained in the newly added subtree. The rotation option computes how many new k-nodes must be added as immediate ancestors of s in order to provide the required number of u-nodes in their other children, when taking the branching factor function into account.

There are many different ways to produce a node insertion plan for a given existing key graph $PS.G$ and a set of joining members. The subtree insertion option can be used to fill the deepest points an existing tree to a desired depth, while the rotation option can be used to increase the height of the tree by adding nodes in shallower locations, increasing the u-node capacity while retaining the tree density (i.e., keeping the tree close to perfect). The Tree scheme produces a plan using the following algorithm:

```
Compute h, the minimum height of a tree that can contain all u-nodes.
while (there are more u-nodes to add) {
    Find s, the shallowest k-node in the tree.
    if (s is a personal key) {
        Find p, the deepest ancestor of s with all descendant u-nodes at the same depth.
        Plan a rotation targeting s to add as many u-nodes as possible while respecting h.
    } else if (all children of s are personal keys) {
        Plan as many subtree insertions of height one (i.e., personal keys) under s as possible.
    } else {
```

```
        Plan as many subtree insertions under s as possible while respecting h.
    }
}
```

Given a node insertion plan, the remainder of the joining procedure can be performed. Figure 10.15 depicts an example of the joining procedure to add 12 joining members to a session with four members. The node insertion plan for the example includes a rotation that replaces an edge with two k-nodes, a rotation that replaces an edge with one k-node, and a subtree insertion of a tree with height two.

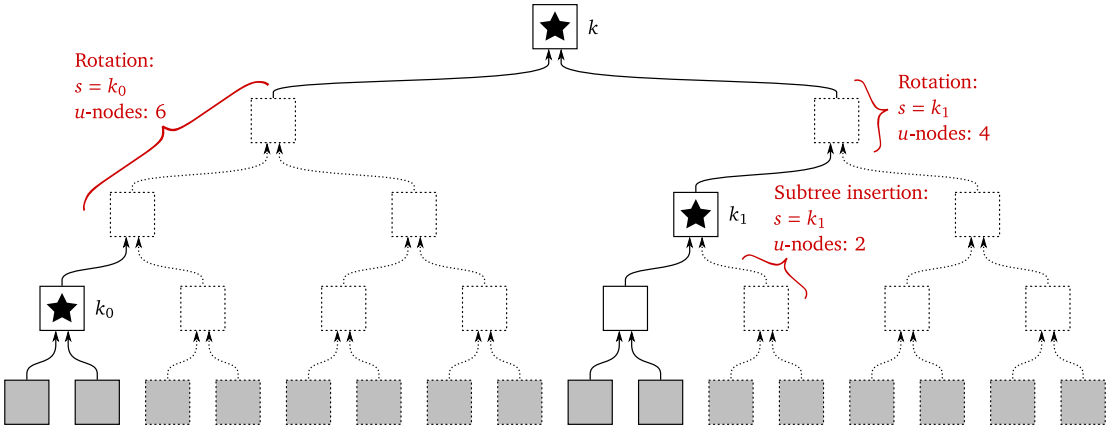
The next step in the joining procedure is to use the node insertion plan to create T^*, the key tree that is instantiated by TKLL as part of KC.PrepareJoin. Figure 10.15a depicts the node insertion plan for the example. In the Tree scheme, T^* is called the *grafting tree*. Transforming the node insertion plan into a grafting tree is straightforward. The node insertion options in the plan are first translated into new subtrees, as described previously. For each of these subtrees, the intended parent k-node in $PS.G$ is added to a set. A topological sort is then performed on this set, producing a subgraph of $PS.G$ that contains the existing root key. A copy of this subgraph is made. For each k-node in the copied subgraph, the scheme keeps track of the original k-node in $PS.G$; this original k-node is said to *correspond* to the k-node in the copy. The newly planned subtrees can then be attached to the copied subgraph at the appropriate points, yielding the grafting tree. Figure 10.15b depicts the grafting tree for the example.

KC.PrepareJoin for the Tree scheme proceeds as follows:

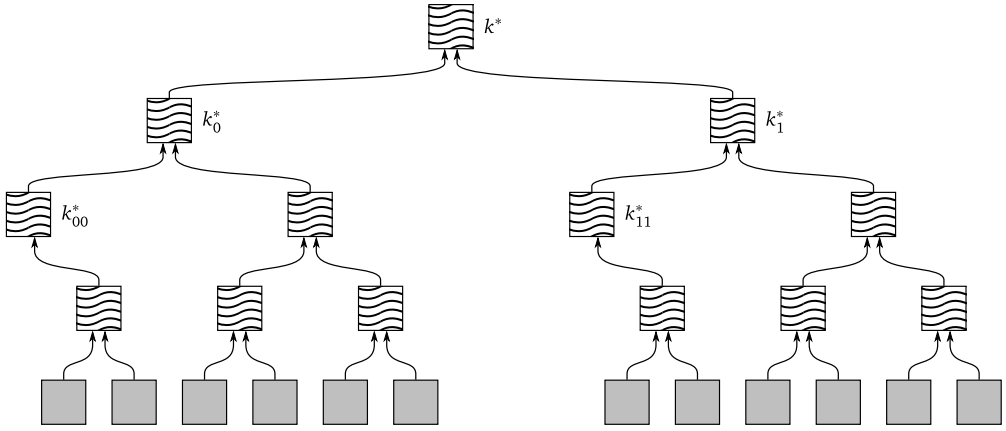
1. Construct the grafting tree, T^*, as described previously.
2. If long-term identities are enabled for the session, add another k-node called bpk^* to T^*, and add an edge from the root of T^* to bpk^*.
3. The joining members use authenticated TKLL to establish key pairs for T^*, using the same definitions for Sig, SVerif, Compute-ID, and Prove-ID as for the Star and Shrub schemes.

The joining commit is produced exactly as for the Star and Shrub schemes: it contains the per-member data for each joining member, the multi-signature σ produced by TKLL, and the public key for each non-personal k-node in T^*, traversed in a deterministic order. The joining private data contains all of the private keys produced by TKLL, Q^*.

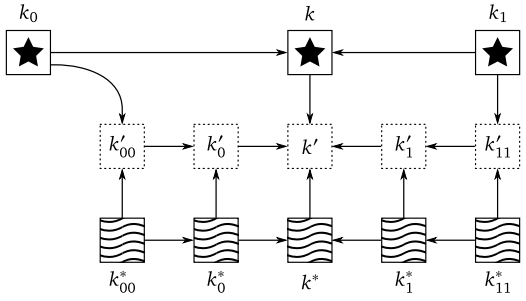
As in the Star and Shrub schemes, KC.MergeJoin begins by verifying the TKLL result and importing the grafting tree with KGM.ImportTree, along with any private keys contained in the joining private data. If long-term identities are enabled, then KGM.ShareKey (see Section 10.5.10)



(a) NODE INSERTION PLAN. The k-nodes with stars are part of the grafting core. New k-nodes are drawn with dashed outlines. This plan uses two rotations and a subtree insertion. The original tree has four non-personal k-nodes.



(b) THE GRAFTING TREE. The optional k-node for bpk^* is not shown.



(c) MERGE OPERATIONS. A fragment of the merged key graph prior to discarding keys.

Figure 10.15 EXAMPLE OF THE JOINING PROCEDURE FOR THE TREE KC SCHEME.
(Refs: 446[abc], 448, and 450[abcde])

is executed to share bpk^* with *rpk*, and then KGM.DiscardKey (see Section 10.5.6) is executed to discard bpk^* from the key graph. Next, a set of *attachment points* is computed: these are k-nodes in the original key graph that were the targets of node insertion options in the node insertion plan. Specifically, the set contains any k-node that was designated as a parent for a subtree insertion option, and any k-node that was designated to become a descendant in the new subtree for a rotation option. Next, a topological sort of the attachment points is performed, yielding a subgraph called the *grafting core*—this is the subset of the original key tree that will be "merged" with the grafting tree. In the example shown in Figure 10.15a, the grafting core consists of the three k-nodes drawn with stars; the two non-root k-nodes in the grafting core are the two attachment points in the example.

Once the grafting tree has been imported into *PS*.*G* and the grafting core has been computed, KC.MergeJoin can begin to merge the two key trees together to achieve the planned result. This is done using the following procedure:

```
for each (k-node k in the grafting core, traversed from shallowest to deepest) {
    if (k has children in the grafting core or was the target of a subtree insertion) {
        Let k* denote the k-node corresponding to k in the imported grafting tree.
        Execute KGM.MergeKeys (see Section 10.5.4) to merge k* and k into a new node k'.
        if (k was not the target of a rotation) {
            Let p denote the parent of k in the original key tree.
            Execute KGM.AddEdge (see Section 10.5.5) to connect k' to p.
        }
    }
    if (k was the target of a rotation) {
        s ← k.
        if (a new node k' was just created for this k) { s ← k' }
        for each (merging node s* being rotated above k, from deepest to shallowest) {
            Execute KGM.MergeKeys to merge s* with s into a new node s'.
            if (s ≠ k) {
                Execute KGM.AddEdge to connect s to s'.
            }
            s ← s'.
        }
        if (k has a parent p in the original key tree) {
            Execute KGM.AddEdge to connect s to p.
        }
    }
```

```
}
for each (k-node k in the grafting core, traversed from shallowest to deepest) {
    if (k was the target of a rotation) {
        for each (merging node s* being rotated above k, from shallowest to deepest) {
            Execute KGM.DiscardKey to discard s*.
        }
    }
    if (k has children in the grafting core or was the target of a subtree insertion) {
        Let k* denote the k-node corresponding to k in the imported grafting tree.
        Execute KGM.DiscardKey to discard k*.
        Execute KGM.DiscardKey to discard k.
    }
}
```

This completes the KC.MergeJoin operation; once the algorithm has been executed, the resulting key graph will be the original key graph with the node insertion plan applied. Note that this algorithm captures an important edge case: if a node is in the grafting core, has no children in the grafting core, is the target of a rotation, and is not the target of subtree insertion, then this node is merged directly with a k-node in the grafting tree corresponding to one of its new parents. In contrast, most nodes in the grafting tree, including targets of rotations that are also parents to other nodes in the grafting core, are merged with a k-node in the grafting tree corresponding to the original node itself. It is important to handle this edge case because the node will have no corresponding node in the grafting tree in this scenario, so the standard approach will not work. Adding a corresponding node to the grafting tree in order to eliminate the edge case would have the side effect of requiring an extra CBRAKEM encapsulation in order to complete the operation, which would harm performance.

To assist with the KC.MergeJoin algorithm, implementations of the Tree scheme may maintain stateful information about the existing tree structure. Primarily, the creation of the node insertion plan can be accelerated by storing, for each k-node in the tree, the node's height above the shallowest descendant with room for additional children (according to the branching factor function), and how many children may be inserted below that descendant. However, the performance costs of the CBRAKEM scheme usually dominate the costs of the tree algorithms, so this additional state is not a critical performance optimization.

The merging algorithm is best visualized as a three-dimensional key graph with the original key tree on one plane and the grafting tree in a parallel plane. The new nodes created by the KGM.MergeKeys executions can be visualized as residing in another parallel plane located between the original tree's plane and the grafting tree's plane. Once the grafting core and

associated nodes in the grafting tree are discarded with KGM.DiscardKey executions, the resulting three-dimensional graph structure can be projected back into a two-dimensional tree on the original plane. While this visualization technique is useful, it is difficult to depict in this document. The example in Figure 10.15 is simple enough to visualize conveniently in two-dimensional space. Figure 10.15c depicts this merging process for the example prior to the key discarding stage, with k-node labels referring to nodes from the grafting core and grafting tree. The arrows depict the edges in the key graph that are created by the executions of KGM.MergeKeys and KGM.AddEdge. The aforementioned three-dimensional visualization technique can be used to interpret the figure in a more generalized way: the original subgraph from the node insertion plan in Figure 10.15a (i.e., ignoring the new nodes with dashed outlines) can be interpreted as the "front" view of a foreground parallel plane, the grafting tree in Figure 10.15b can be interpreted as the "front" view of a background parallel plane, and the merged graph in Figure 10.15c can be interpreted as the "top" view (not to scale) of the three parallel planes, with the new nodes in the middle parallel plane. Because of the simplicity of the example, the merged graph can be projected into two dimensions without overlapping nodes, but this is not the case in general. Note that while this geometric interpretation may aid with intuitively understanding the effects of the algorithm, actual implementations of the Tree scheme would simply execute the previously described graph algorithms without this interpretation.

10.11.5 Choosing a Key Control Scheme

The developer must select a KC scheme and its parameters when deploying a secure group messaging protocol based on Safehouse. This chapter described several possibilities at a high level. Aside from implementation complexity, choosing between the schemes is largely a matter of estimating the intended number of members in a session.

The Trivial (see Section 10.11.1), Star (see Section 10.11.2), Shrub (see Section 10.11.3), and Tree (see Section 10.11.4) schemes ultimately represent points on a tradeoff spectrum. Simpler schemes, like the Star scheme, add very few non-personal k-nodes to the key graph (none in the case of the Trivial scheme). Complex schemes, like the Tree scheme, add many non-personal k-nodes. In general, schemes with more non-personal k-nodes make KGM.Evict (Section 10.5.7) and KC.MergeJoin perform more CBRAKEM encapsulations to fewer recipients. The exact increase in encapsulations and decrease in recipient counts depends on the shape of the key graph. In the case of the Tree scheme, higher trees require more CBRAKEM encapsulations, and larger branching factors require larger recipient sets for those encapsulations. The branching factor function controls the tradeoff between the height and width of the tree for a given group size. In the case of the Shrub scheme, increasing the maximum set size also increases the number

of recipients when using CBRAKEM to replace a set key, but it reduces the number of recipients when using CBRAKEM to replace the root key. Selecting optimal parameters for the Shrub and Tree schemes depends on the exact performance costs of the CBRAKEM scheme for various encapsulation and recipient counts, which was evaluated empirically in Section 8.6, and the expected number of members in a session. The cost of executing KGM.Evict is the main cost of executing MLS.MassEvict (see Section 10.10.5), MLS.PCSRatchet (see Section 10.10.7), and MLS.Cleanup (see Section 10.10.8). Optimizing MLS.PCSRatchet is particularly important when it is in use by the developer, because the frequency with which it is executed scales linearly with the number of members in the group.

In general, simpler schemes (e.g., Trivial, Star, and Shrub) perform better for small groups, and more complex schemes (e.g., Tree) perform better for large groups. An empirical evaluation and comparison of key control schemes is presented in Section 10.13, providing specific insight into the best schemes to choose for various scenarios.

10.12 Steady-State Behavior

The developer interacts with Safehouse by using the serialization functions described in Section 10.9.2, the signing functions described in Section 10.9.3, and the MLS-layer operations described in Section 10.10. This section describes how this interface should actually be used in a secure group messaging protocol.

Initially, a Safehouse session is created by a single group member. This member creates a commit that performs MLS.Initialize (see Section 10.10.2). The developer may also want the member to set certain labels in the public or confidential label-value stores using the appropriate MLS-layer operations (see Section 10.10.12). It is possible to enforce that labels (either public or confidential) and public values are set during the initialization by configuring an appropriate group policy that would otherwise deny the transaction. Similarly, the group policy can enforce the creation of a bearer invitation with unlimited uses during initialization in order to implement an "open" group, as discussed in Section 10.1.9.

Once a Safehouse session is initialized, new members can join the session. The server can queue and aggregate new members, inform them that they are permitted to join the group, distribute their agent IDs and personal public keys, and relay interactive messages between them as they execute KC.PrepareJoin (see Section 10.11). Once the protocol is complete, a representative of the new members can use the joining commit and its joining private data to perform an MLS.MassJoin operation (see Section 10.10.4). The resulting commit incorporates all of the new members into the session. As with the initialization procedure, the group policy

can be configured to enforce that the representative also sets values in the public or confidential label-value store as part of the commit containing the MLS.MassJoin operation.

Members in a session can store their group state on non-volatile storage devices using the serialization functions (see Section 10.9.2). However, as discussed in Section 10.1.1, members should automatically securely erase their private state if they have been unable to connect to the session for longer than the forward secrecy period. Members that have been disconnected for an entire forward secrecy period must rejoin the group again after connectivity has been restored in order to continue participation.

There are several MLS-layer operations that are typically performed in commits in response to end-user action: issuing or retracting invitations (see Section 10.10.3), evicting members (see Section 10.10.5), changing the developer mode (see Section 10.10.11), and altering the label-value stores (see Section 10.10.12). The developer might also choose to automatically perform commits with these operations. For example, some secure group messaging tools might automatically issue invitations in reaction to events occurring in external protocols.

There are several operations that should always be performed automatically by members in a Safehouse session. Tools should periodically check if any of these operations need to be performed. A Safehouse library might silently implement these automated operations and hide them from the developer's group policy, ensuring that all agents accept these necessary maintenance operations. The following maintenance operations should be performed automatically:

- MLS.RetractInvitation (see Section 10.10.3.3) should automatically remove any expired layaways; this decreases the overhead of encrypting blinded identities in the identity table and anchor keys in the label-value stores.
- MLS.FSRatchet (see Section 10.10.6) should be executed frequently on a per-session (not per-member) basis in order to provide stronger forward secrecy. For example, a member might automatically perform an MLS.FSRatchet commit if none was performed by any member during the last 24 hours. When Safehouse is used in secure group messaging protocols that enforce a total ordering on payloads and the payloads are infrequent, it is plausible to enforce that a commit containing an MLS.FSRatchet operation is performed alongside every payload.
- If post-compromise security is enabled, then members must perform MLS.PCSRatchet (see Section 10.10.7), and may optionally perform MLS.PCSReidentify (see Section 10.10.9), at least once during each forward secrecy period. This ensures that any previously compromised private keys known to the member can no longer be used to decrypt payloads or future changes to the label-value stores.

- If post-compromise security is disabled *and* the k-nodes on the path to a member's u-node are the target of superfluous edges, then the member must perform MLS.Cleanup (see Section 10.10.8) at least once during each forward secrecy period. This ensures that MLS.MassEvict (see Section 10.10.5) remains efficient by reducing the prevalence of superfluous edges.

- Members automatically authenticate to new members by performing MLS.Authenticate operations (see Section 10.10.10).

- Members that fail to perform the required automated tasks can be automatically evicted from session using an MLS.MassEvict operation. Each agent (including the server) can maintain local state about the last time that members performed their required automated tasks and automatically evict them after too much time has passed. Agents will only accept these automated evictions if their own local state also shows that too much time has passed. It is safe for members that have recently joined the group and have no record of the time since other members last performed operations to blindly accept these automated evictions: malicious evictions using this mechanism would require cooperation from a malicious server and would fork the group, which a malicious server can do at any time in any case.

There is no built-in mechanism in Safehouse for a member to unilaterally "leave" a session, because there is no mechanism for a leaving member to correctly replace private keys known to them with ones that are unknown to them in a publicly verifiable manner—BRAKEM always reveals the new private keys to the agent performing the encapsulation. However, there are several ways to achieve the same effect in practice. The simplest solution is for the "leaving" member to delete their group state. This will cause other members to automatically evict the leaving member from the group after the automated duties are not performed within the forward secrecy period. To avoid the need to wait for the entire forward secrecy period, the overall secure group messaging protocol could implement functionality that allows the leaving member to signal their intention (e.g., by adding an entry to the public label-value store authorizing an eviction), allowing another member to evict them from the session. The group policy could be configured to recognize this signaling mechanism and allow the associated commits. Client software could also be configured to automatically evict leaving members—potentially as a precondition before sending new messages. User interfaces could indicate that a member is in the process of leaving (indicating that they have deleted their private state, assuming that they are behaving honestly) for the duration between the receipt of the leaving signal and the receipt of the commit that actually evicts the member.

10.13 Performance Evaluation

A prototype implementation of Safehouse—including BRAKEM$^{\text{DDL}}_{\star}$, BRAKEM$^{\text{ZK}}$, TKLL, and all of the functions and operations described in this chapter—was developed as part of this work. An empirical evaluation of this prototype was performed in order to answer two questions about Safehouse's performance:

1. What is the cost, in terms of time and space, to secure a realistic messaging protocol using Safehouse, and is this cost reasonable?
2. Of the key control schemes introduced in Section 10.11, which schemes are the best candidates for sessions of different sizes?

In the experiment, the prototype implementation was used to replay a collection of conversation transcripts as if they had been secured using Safehouse. The experiment evaluated several important metrics for each conversation, including the communication overhead and the computational overhead for all agents.

In terms of the MLS operations presented in Section 10.10 and the steady-state behavior described in Section 10.12, the experiment included only a subset of the available functionality. Each session included only MLS.MassJoin commits, MLS.MassEvict commits, either MLS.Cleanup or MLS.PCSRatchet commits, and payload transmissions. The experiment did not include any operations related to invitation or label-value store management: these operations are optional and their use is situational. Each session in the experiment included a single bearer invitation with unlimited uses in order for new members to join; this is the technique suggested in Section 10.1.9 to implement "open" groups.

The remainder of this section is organized as follows: Section 10.13.1 describes how the conversation transcripts used in the experiment were produced, Section 10.13.2 discusses how to estimate the performance overhead of the scheme on consumer hardware, Section 10.13.3 describes the approach used in the experiment to emulate Safehouse sessions, Section 10.13.4 presents and discusses the results, and Section 10.13.5 summarizes the findings into actionable deployment advice.

10.13.1 Data Sets

To emulate a Safehouse session, the experiment required a data set that specified a sequence of events, including the virtual timestamps indicating when each event occurs, with the following possible event types:

- **Mass join**: a set of new members with given agent IDs joins the session.
- **Mass evict**: a set of existing members with given agent IDs are evicted from the session.
- **Payload**: an existing member with a given agent ID sends a payload with a given plaintext size.

The experiment involved emulating multiple Safehouse sessions for several different data sets. Although Safehouse allows agent IDs to be reused over the lifetime of the session, for convenience, each data set was expected to include at most one join and evict event for each agent ID. Additionally, each data set only needed to include the *sizes* of payload plaintexts and not the plaintexts themselves, since the IND-CCA2 cryptosystem used to encrypt the payloads would exhibit the same behavior for any plaintexts of the same size. Given a data set containing these types of events with timestamps, the experiment automatically determined the virtual times at which each group member should perform an MLS.Cleanup or MLS.PCSRatchet operation. Data sets with this form are also general enough to be used in experiments evaluating alternative secure group messaging protocols. To achieve the desired generality, the "mass evict" events in the data sets did not specify which member performed the eviction, because secure group messaging protocols other than Safehouse may not necessarily require an unevicted member to participate in an eviction.

There were several possible options for acquiring data sets to use in the experiment. One option was to randomly generate a *synthetic* data set: an invented sequence of events that satisfies some desirable characteristics, such as the average number of members in the group. Synthetic data is useful for evaluating a spectrum of scenarios with carefully controlled properties, but these scenarios might be different from scenarios encountered in practice. Another option was to use a real data set: a sequence of operations and payloads that was observed in a real target scenario. Real data sets can provide stronger evidence that the results will hold in practice, but relatively few real data sets are available.

Unfortunately, none of the real data sets that were available contained enough data to produce data sets with the aforementioned structure. There are many sources of publicly logged IRC conversations, but the IRC protocol does not support non-interactivity. Consequently, IRC clients frequently disconnect and reconnect as their network connections are temporarily interrupted.

Moreover, many public IRC logs do not include records of these connection and disconnection events, nor the set of members in the channel. Most other publicly available data sets involved protocols that inherently lacked the concept of a stateful conversation, such as social media posts, emails, or SMS message logs. The most widely used protocols that provide both non-interactivity and stateful conversations, such as Discord and Slack, are generally used in private deployments that do not publish public logs.

As a compromise, the experiment used several *partially synthetic* data sets: combinations of data contained in real data sets with additional synthetic data that achieved the required structure. Publicly available IRC logs were used as a source of payloads. The IRC logs defined the sequence of messages that were sent, the usernames (known as *nicknames* in the IRC protocol) that sent them, and the associated timestamps. Mass join and mass evict events were then synthetically generated for every payload sender appearing in the data set. These events were added based on the assumption that, if IRC was replaced by a protocol like Safehouse that supports non-interactive communication, then the senders appearing in the data set would remain in the session for as long as possible.

The Ubuntu Chat Corpus [UA13] supplied the real chat logs that served as the basis of the data sets used in the experiment. This corpus consists of IRC logs of public technical support channels for users of Ubuntu, a popular Linux-based operating system. The corpus contains messages sent to 11 different channels between 2004-05-07 and 2012-11-30.[34] Importantly, all of the users appearing in the logs were informed that their messages would be released publicly. The log for each channel is a text file containing nickname changes and various types of messages, as they would be displayed in a typical IRC client; it does not contain notifications of users joining or leaving the channel. The log includes the date and time that each message was sent, expressed as the hour and minute—logs do not contain the second or timezone. Compared to the target scenarios described in Chapter 7, most users in technical support channels are expected to participate in the session for relatively short periods of time—just long enough to receive technical support. For this reason, emulating these chat logs using a secure group messaging protocol is particularly expensive.

To construct the data set for the experiment, three channels were selected from the Ubuntu Chat corpus: `#ubuntu-se`, `#ubuntu-cn`, and `#ubuntu`. These channels are technical support forums with messages written in Swedish, Chinese, and English, respectively. These channels were selected because they represent “small”, “medium”, and “large” group conversations. All data prior to 2008 was discarded because earlier data in the `#ubuntu` channel log contains ambiguous timestamps (12-hour times with no a.m. or p.m. markers) and nicknames with special

[34] More recent public logs for the same channels are available, but they are not as easily accessible as those in the Ubuntu Chat Corpus, and the number of users participating in the channels has slowly decreased over time.

characters that cannot generally be distinguished from message contents. The `#ubuntu-se` and `#ubuntu-cn` channel logs begin in 2010, so no data was discarded from these logs. Next, network-wide messages and private messages to the logging bot were discarded, leaving only messages that were sent to the channels. The messages in each channel were then processed chronologically and sent through a series of processing layers, producing a data set consisting of the desired event types as output. Messages in the original chat logs were substituted with "payload" events that contain the size of the message (in terms of bytes appearing in the original chat log), but not the original message itself—as discussed previously, message contents were irrelevant for the experiment.

The first processing layer tracked nicknames participating in the conversation at every point in time. When a message from a new nickname was encountered, a new agent ID was assigned to the nickname, and a "join" event was inserted into the output immediately preceding the message. When a nickname change was encountered, the agent ID was unassigned from the previous nickname and assigned to the new one. As a special case, if the logs contained a nickname change to a nickname that was already active in the conversation, then the identities were "merged"—this typically occurs in IRC when a user rejoins under an alternate nickname after a connection interruption and then automatically restores their nickname once the original connection has timed out. To merge agent IDs, all events that were previously output using the agent ID assigned to the old nickname were rewritten to use the agent ID assigned to the new nickname, and the "join" event that occurred later was removed. After processing each message, "evict" events were added to the output for all agent IDs in the session that last sent a message over a week ago (in terms of the timestamps in the log). "Evict" events were given a timestamp of exactly 24 hours after the last message received from the user.

The second processing layer aggregated "join" and "evict" events into "mass join" and "mass evict" events. The data set was processed so that any "join" and "evict" events occurring within sliding 5-minute windows would be combined into aggregated events. For example, if users with agent IDs 1–6 joined the conversation at 00:01, 00:02, 00:05, 00:08, 00:10, and 00:12, respectively, then the events would be aggregated into a "mass join" of agent IDs 1–3 at 00:01 and a "mass join" of agent IDs 4–6 at 00:08. Similarly, if the users left the conversation at 01:01, 01:05, 01:14, 01:16, 01:16, and 01:18, respectively, then the events would be aggregated into a "mass evict" of agent IDs 1 and 2 at 01:05, and a "mass evict" of agent IDs 2–6 at 01:18.

Finally, the third processing layer removed any messages sent to the channel when there was only one active participant. This can occur in the Ubuntu Chat Corpus because users that "idled" in the channels have no presence in the logs. However, for a data set intended for performing experiments with secure group messaging protocols, it was convenient to ensure that all messages appearing in the data set actually represented communication between clients.

Table 10.1 Properties of the data sets used to evaluate the Safehouse prototype. (Refs: 458 [a b c], 463, 464, 466, 467, 469, 471, and 475)

	`#ubuntu-se`		`#ubuntu-cn`		`#ubuntu`	
First event	2010-11-04		2010-11-04		2008-01-01	
Last event	2012-11-30		2012-11-30		2012-11-30	
Measurements start	2010-11-12		2010-11-12		2008-01-08	
Measurements end	2012-11-29		2012-11-29		2008-02-07	
Messages / day	720	(380)	2 140	(770)	10 990	(900)
Payload bytes / day	35 000	(19 000)	88 000	(30 000)	691 000	(58 000)
Mass joins / day	6.5	(3.1)	22.9	(6.4)	191	(5.9)
Joining members / day	6.7	(3.3)	26.3	(8.4)	501	(30)
Mass evicts / day	5.8	(2.7)	21.9	(6.6)	190	(6.7)
Evicted members / day	6.7	(3.3)	26.3	(8.5)	502	(29)
Members	52	(12)	127	(28)	1386	(46)

The measurements are means with standard deviations in parentheses. The means were computed based on all events with dates in the "measurements start" to "measurements end" range, inclusive. The mean number of members in the group was computed on a per-second basis.

The data sets for the three channels produced using this procedure were stored as `gzip`-compressed text files with one JSON-encoded event per line. This format makes it easy for other researchers to use the same data sets to test alternative secure group messaging protocols. Table 10.1 presents various measurements of the events contained in the data sets. For all three data sets, the experiment involved emulating the conversation beginning with the first event in the log. However, since "evict" events were generated for participants that did not send any messages for at least one week, the data sets began with a "warming up" phase in which there were far more "join" events than "evict" events. Consequently, the data collected in the experiment was only processed for the time period beginning one week (in virtual time) after the first event in the data set. The "measurements start" and "measurements end" rows in Table 10.1 indicate the first and last dates in the data sets for which experimental results were collected. Although the data set for the `#ubuntu` channel contained almost five years of events, the experiment only measured at most one month of the emulated session due to computational constraints. The measurements in Table 10.1 show that the number of members in each session remained relatively constant during the measurement period—the "joining members / day" and "evicted members / day" are essentially equal within each of the three data sets.

10.13.2 Estimating BRAKEM Costs

In order to emulate Safehouse sessions for the data sets described in Section 10.13.1 in a reasonable amount of time, the experiment had to be performed on a large machine in an academic computing cluster. The emulation was expensive because each commit induced by the data set had to be applied by every member in the session, and all of these applications had to be emulated. One of the downsides of using a computing cluster to perform the experiment is that time measurements do not reflect the expected performance of consumer devices. Consequently, the performance of Safehouse in terms of computation time had to be evaluated indirectly from the experimental measurements.

The computation time required to execute the Safehouse protocol is dominated by the underlying BRAKEM scheme, which requires many modular exponentiations for encapsulation, decapsulation, and verification. This means that it is possible to derive a good approximation of the computation time required to execute a session on a consumer device in the following manner:

1. Emulate Safehouse sessions for the data sets in the computing cluster and record the number of BRAKEM encapsulations during the performance of each operation, the number of rings (i.e., sets of recipients) in each encapsulation, and the number of recipients in each ring.

2. On a consumer device, measure the time required to execute the BRAKEM functions for encapsulations with different *shapes*: different numbers of rings and recipients.

3. Derive a statistical model to extrapolate the time required to execute the BRAKEM functions on the consumer device for encapsulations of any shape.

4. Apply the model to the encapsulation shapes recorded during the emulation to derive an approximation of the total time requirements if the session had been executed on the consumer device.

Step 2 is essentially the performance evaluation described in the context of comparing BRAKEM constructions in Section 8.6, except that the evaluation must be performed for a larger set of possible encapsulation shapes. As part of the experiment, the performance of BRAKEM.Encapsulate, BRAKEM.Decapsulate, and BRAKEM.Verify was evaluated 100 times using all Skylake cores on an Intel Core i7-6700K CPU with all core frequencies pinned to 4.0 GHz and Intel Turbo Boost disabled; this was the same consumer machine that was used in the evaluation of the new DAKEs in Section 5.9 and in all of the performance evaluations of BRAKEM in Section 8.3.2. The experiment measured the BRAKEM time for shapes with 1–10 rings and

Table 10.2 A PERFORMANCE EVALUATION OF BRAKEM$^{\text{DDL}}_{\star}$.DECAPSULATE FOR VARIOUS SHAPES. All times are means given in milliseconds. Standard deviations are not shown, but they are between 2 and 5 milliseconds for all values. (Refs: 460 [a] [b])

		Recipients / ring									
		1	2	3	4	5	6	7	8	9	10
Rings	1	13	13	14	14	14	14	14	14	14	14
	2	18	18	19	18	19	19	19	19	19	19
	3	26	26	26	27	27	27	28	27	28	27
	4	31	31	32	31	32	32	32	32	32	33
	5	34	35	36	37	37	37	37	37	37	38
	6	39	39	40	41	41	41	42	42	42	42
	7	42	43	44	45	45	46	46	46	46	47
	8	47	48	48	49	50	50	50	50	51	51
	9	50	51	52	53	54	54	55	56	55	56
	10	54	55	57	58	58	58	59	60	60	60

1–10 recipients in each ring, where all rings had the same number of participants in each scenario. The experiment was repeated for the prototype implementations of both BRAKEM$^{\text{DDL}}_{\star}$ and BRAKEM$^{\text{ZK}}$. As an illustrative example of this intermediate step, Table 10.2 depicts the results of the experiment for BRAKEM$^{\text{DDL}}_{\star}$.Decapsulate. The results for the other two BRAKEM$^{\text{DDL}}_{\star}$ functions and the three BRAKEM$^{\text{ZK}}$ functions are not shown.

For each of the six BRAKEM functions (three functions for each of the two constructions), the experiment produced a two-dimensional matrix of mean timing measurements, such as the results for BRAKEM$^{\text{DDL}}_{\star}$.Decapsulate in Table 10.2. Consider how the shape of the encapsulation affects the the work that is performed by these functions. Both BRAKEM constructions follow the general design described in Section 8.1.1. For all functions, this design results in a fixed cost for each ring, and a different fixed cost for each recipient in each ring. In terms of BRAKEM$^{\text{DDL}}_{\star}$ (see Section 8.2.8): for each ring i, the machine generates or processes pk^*_i, sk^*_i, r_i, y_i, h_i, k_i, and π_i; and for each recipient j in each ring i, the machine generates or processes $h_{i,j}$. In other words, the per-ring set of operations scales linearly with the number of rings, and the per-recipient set of operations scales linearly with the number of recipients in each ring. Additionally, in the case of BRAKEM$^{\text{DDL}}_{\star}$, the proof statement for Π_0 grows to contain a term for each recipient in each ring. Careful evaluation of the proof system described in Section 8.2.7 shows that this also induces linear scaling with respect to the same shape parameters. A similar analysis can be performed for BRAKEM$^{\text{ZK}}$ (see Section 8.5.5).

Table 10.3 Models for the BRAKEM performance of the consumer machine. x is the number of rings, and y is the total number of recipients (across all rings). The models output time values in milliseconds. The coefficients are given with four significant figures because x and y are always less than 2 000 for the scenarios of interest. R^2 is the coefficient of determination with respect to the empirically measured means used to establish the linear regression.
(Refs: 461, 464, 465, 467, 468, and 469)

	Predicted mean time [ms]	R^2
$\text{BRAKEM}_{\star}^{\text{DDL}}$.Encapsulate	$0.4776 + 4.079x + 0.2451y$	0.9998
$\text{BRAKEM}_{\star}^{\text{DDL}}$.Decapsulate	$10.92 + 4.472x + 0.06165y$	0.9874
$\text{BRAKEM}_{\star}^{\text{DDL}}$.Verify	$10.97 + 3.667x + 0.05584y$	0.9827
$\text{BRAKEM}^{\text{ZK}}$.Encapsulate	$77.88 + 133.9x + 0.5260y$	0.9969
$\text{BRAKEM}^{\text{ZK}}$.Decapsulate	$2.083 + 0.6338x + 0.2051y$	0.9744
$\text{BRAKEM}^{\text{ZK}}$.Verify	$2.109 + 0.5038x + 0.2080y$	0.9723

Based on the aforementioned analysis of the implementation, the results of the experiment should closely match a model that predicts time requirements based on a linear equation in terms of the number of rings and the total number of recipients. The `scikit-learn` Python library[35] was used to compute linear regressions for all six functions based on the experimental results. The linear regressions accurately predict the results of the experiment with very high coefficients of determination. The resulting models are presented in Table 10.3. These models can be used to predict the mean performance of the i7-6700K machine when performing or applying the commits in the emulated sessions.

10.13.3 Session Emulator

The session emulator that was developed for the experiment read the data sets described in Section 10.13.1, emulated Safehouse sessions that performed the specified sequences of operations, and outputted performance measurements to an SQLite database for each emulated session. Each execution of the session emulator was parameterized with the KC scheme to use and whether to use MLS.Cleanup or MLS.PCSRatchet operations. All sessions were instantiated using the $\text{BRAKEM}_{\star}^{\text{DDL}}$ construction and were configured for ephemeral authentication.

The session emulator was designed to run on a large machine in an academic computing cluster. The experiments were performed on a machine with eight Intel Xeon E7-8870 processors; each of these processors has 10 physical Westmere-EX cores running at 2.4 GHz and with hyperthreading enabled. To take advantage of this configuration, the session emulator was

[35] https://scikit-learn.org/

designed to exploit massive parallelism while being aware of the NUMA topology (i.e., ensuring that processes did not share memory across different processors). For this reason, the session emulator spawned one worker process for each active member in the session, allowing the Linux scheduler, rather than the Go runtime, to handle most of the scheduling work. These worker processes were recycled as existing members were evicted and new members joined. Each of these worker processes was pinned to a single processor using `taskset`. The master process was responsible for reading the input data set and managing worker processes. The master process used the `stdin` and `stdout` pipes for each worker process to assign the active agent ID, instruct them to perform Safehouse operations, or to receive a commit from the server and apply it. The master process also implemented a simplified Safehouse server that would receive commits, apply them to the server state, and relay them to all other clients. The worker processes connected to this server using `localhost` TCP connections; all active members in the session remained connected to the server for the entire duration of their participation in the session, avoiding the need to implement a storage mechanism for commits. A similar `localhost` server was also provided by the master process to relay transmissions between joining members during execution of the KC.PrepareJoin protocol. As an important optimization, the large precomputed modular exponentiation tables for $\text{BRAKEM}_{\star}^{\text{DDL}}$ discussed in Section 8.3.2.5 were computed only once and shared between all worker processes by using an anonymous in-memory file initialized with `memfd_create`.

The data sets described in Section 10.13.1 did not explicitly specify when to perform MLS.Cleanup (or MLS.PCSRatchet) operations, or the existing members responsible for performing MLS.MassEvict operations. Omitting these instructions from the data sets ensured that they were generalized enough to be used in evaluations of alternative secure group messaging protocols. The session emulator tracked the last cleanup time of each member in the session and instructed the associated worker process to perform an MLS.Cleanup or MLS.PCSRatchet operation (based on the configuration of the experiment) every week (in terms of virtual time as specified in the data set). When instructed to perform a mass eviction, the session emulator selected an existing, unevicted member uniformly at random to perform the commit. During the application of commits to the server state, the master process recorded a variety of measurements and stored them in the SQLite database:

- a description of the operation that was performed;
- the size of the resulting commit, in bytes;
- a list of all BRAKEM encapsulations performed as part of the commit, the number of rings in each encapsulation, and the number of recipients in each ring;

- for "mass join" operations, the total number of bytes transmitted between joining members during the KC.PrepareJoin protocol;
- the size of the public state, in bytes, before and after the application of the commit; and
- the number of members in the session before and after the application of the commit.

The session emulator did not actually emulate the transmission of payloads, because they would not affect the group state in the worker processes. Instead, when the master process encountered a payload in the data set, it recorded the size of the message and the size of the digital signature that the sender would attach in practice—for $\text{BRAKEM}_{\star}^{\text{DDL}}$, the signature is always 64 bytes. The session emulator did not record the cryptographic overhead of the IND-CCA2 encryption for payloads because Safehouse does not impose a choice of cryptosystem.

10.13.4 Results

In the experiment, the session emulator measured Safehouse sessions for the `#ubuntu-se`, `#ubuntu-cn`, and `#ubuntu` data sets (see Table 10.1) using five different KC schemes with MLS.Cleanup:

1. the Trivial scheme (see Section 10.11.1);
2. the Star scheme (see Section 10.11.2);
3. the Shrub scheme (see Section 10.11.3) with maximum set size $n_S = 5$;
4. the Tree scheme (see Section 10.11.4) with a constant branching factor function $bf(d) = 10$; and
5. the Tree scheme with a constant branching factor function $bf(d) = 64$.

Due to computational constraints, the `#ubuntu` data set was only emulated using the Trivial, Star, and Shrub schemes for one week (in virtual time) instead of one month—this is sufficient data to demonstrate that these schemes are drastically inferior to the Tree scheme for large groups. To understand the overhead of using MLS.PCSRatchet, the experiment also emulated a session for `#ubuntu-se` using the Tree scheme with $bf(d) = 10$ and MLS.PCSRatchet instead of MLS.Cleanup. In total, the experiment involved 16 scenarios. The number of scenarios was limited because the purpose of the experiment was to gather preliminary intuition about the practicality and scalability of potential KC schemes; developing superior algorithms and finding

optimal configurations is left as future work. The $n_S = 5$ and $bf(d) = 10$ configurations for the KC schemes were chosen arbitrarily to test the schemes. The choice of $bf(d) = 64$ for the second Tree configuration was based on the coefficients in the $\text{BRAKEM}_{\star}^{\text{DDL}}$.Verify performance model (see Table 10.3), rounded down to the nearby power of 2 for convenience: $3.667/0.05584 \approx 2^6$.

The per-commit measurements produced by the session emulator, as described in Section 10.13.3, were processed to derive aggregated statistics, with the goal of answering the experimental questions (see the start of Section 10.13). In order to capture the behavior after the initial "warming up" phase, only measurements for operations performed between the "measurements start" and "measurements end" date ranges listed in Table 10.1 were considered. The following statistics related to communication and storage costs, expressed as means and standard deviations, were computed for each of the 16 scenarios:

- the average daily size of the commits;
- the average daily size of the digital signatures attached to payloads;
- the average daily size of the AEAD tags attached to payloads,[36] assuming that each encryption produces a typical 128-bit authentication tag;
- the average daily size of the transmissions for KC.PrepareJoin protocol executions; and
- the average size of the public state, on a per-second basis.

Reporting a measurement as only a mean and standard deviation can hide outlier samples that may have significant real-world impact (e.g., unrealistic storage requirements). This is not the case for the measurements in this experiment: daily commit size and KC.PrepareJoin transmission size is based on the number of members in the group and the number of group membership changes, daily signature and AEAD tag sizes are based on the number of message transmissions, and the public state size is based on the number of members in the group. Without a sudden increase in the number of members, membership turnover, or message frequency, there will not be a sudden increase in any of the measurements.

[36] ^ AEADs also require a (public) nonce to encrypt or decrypt payloads. In most applications, the developer can choose an AEAD designed to work with sequentially selected nonces, such as ChaCha20-Poly1305 [NL18]. Each member of the group can use a KDF to derive an encryption key from their agent ID and the group key. By using this personalized key for all payloads, each member can safely use sequential nonces for its outgoing payloads. This approach allows the nonces to be encoded using a variable-length integer encoding, such as the one used by Google's protocol buffers [Goo20], making the transmission cost of nonces negligible. If this approach is not feasible for an application and 128-bit random nonces must be attached to payloads, then the added transmission cost will be identical to the evaluated transmission cost of the 128-bit AEAD tags.

The BRAKEM shape information recorded by the session emulator was supplied to the performance models in Table 10.3 in order to estimate computation time requirements for the consumer device to act as a member in the session. The computation time estimates were produced for both $\text{BRAKEM}_{\star}^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$, even though the communication and storage costs were directly measured for only $\text{BRAKEM}_{\star}^{\text{DDL}}$; an evaluation of the communication costs of the session when instantiated with $\text{BRAKEM}^{\text{ZK}}$ is left to future work. For each member that participated in the session, the experimental results were searched to find all of the days in which that member performed at least one operation. The BRAKEM shape information for these operations was provided as input to the performance models; the result was the estimated amount of time per day spent by the member to perform commits, excluding days in which the member did not perform any commits. These calculations were then averaged across all members in the session to produce the "daily BRAKEM Encapsulate / performer" metric.

For each day in the experimental results, all of the BRAKEM shapes were provided as input to the performance models. The "daily BRAKEM max Decapsulate / member" metric is the sum of the estimated cost to call BRAKEM.Decapsulate on all of the shapes, averaged across all days. In a real deployment, members generally do not decapsulate all of the BRAKEM transfers, because usually there are a few transfers for which the member is not an intended recipient. Consequently, this metric serves as an upper bound for the amount of time that a member would spend applying all of the commits for a day. Similarly, the "daily BRAKEM Verify" metric is the sum of the estimated cost to call BRAKEM.Verify on all of the shapes, averaged across all days. This metric is the estimated time that the server will spend applying the commits for a day, because the server never decapsulates a BRAKEM transfer. It also serves as a lower bound for the time spent by a member to apply the commits. These upper and lower bounds are generally quite close in the experimental results, because decapsulation is only marginally more expensive than verification for both $\text{BRAKEM}_{\star}^{\text{DDL}}$ and $\text{BRAKEM}^{\text{ZK}}$.

The communication sizes and computational requirements are presented in the experimental results as daily averages. Safehouse is designed to support non-interactive communication, so members may be disconnected for long periods of time (within the forward secrecy period) before reconnecting to the server. Upon reconnection, the member will receive and apply the commits that were performed while it was disconnected. The daily averages in the results provide an estimation of how much commit data will accumulate during these periods and how long it will take for the member to apply the commits to its locally stored group state.

10.13.4.1 Small Group: #ubuntu-se

Table 10.4 presents the results of the experiment for the `#ubuntu-se` data set, representing a "small" group. The results suggest that securing an IRC channel like `#ubuntu-se` requires

Table 10.4 PERFORMANCE EVALUATION OF SAFEHOUSE FOR A SMALL GROUP. This table shows the experimental results for emulating the `#ubuntu-se` data set. The Trivial scheme generally performed the best for this data set. The Tree scheme with $bf(d) = 64$ essentially behaves like the Star scheme with less efficient public state compression, because the average number of members (52, according to Table 10.1) is smaller than the branching factor. (Refs: 465, 470, and 474 [a] [b])

	Trivial	Star	Shrub $n_S = 5$	Tree $bf(d) = 10$	Tree $bf(d) = 64$
Daily update data:					
Commits [KiB]	550 (250)	570 (230)	740 (270)	730 (270)	560 (220)
PrepareJoin messages [KiB]	2.8 (1.4)	3.2 (1.7)	5.6 (2.8)	5.6 (2.8)	3.2 (1.9)
Daily payloads (common):					
Plaintext (goodput) [KiB]	.....		34 (18)	.....	
Signatures [KiB]	.....		45 (24)	.....	
AEAD tags [KiB]	.....		11.3 (6.0)	.....	
Storage (per-second average):					
Public state [KiB]	36.2 (4.4)	36.6 (4.4)	42.5 (4.8)	40.4 (4.4)	37.1 (4.7)
Daily computation:					
Perform probability / member	0.19	0.19	0.22	0.22	0.19
BRAKEM$^{\text{DDL}}_{\star}$ total time / day:					
Encapsulate / performer [s]	0.022 (0.006)	0.019 (0.007)	0.019 (0.006)	0.019 (0.006)	0.018 (0.006)
Max Decapsulate / member [s]	0.33 (0.13)	0.36 (0.14)	0.47 (0.17)	0.46 (0.17)	0.36 (0.14)
Verify (for server) [s]	0.32 (0.13)	0.34 (0.13)	0.42 (0.16)	0.42 (0.16)	0.34 (0.13)
BRAKEM$^{\text{ZK}}$ total time / day:					
Encapsulate / performer [s]	0.37 (0.12)	0.50 (0.13)	0.64 (0.20)	0.63 (0.20)	0.50 (0.13)
Max Decapsulate / member [s]	0.191 (0.093)	0.130 (0.060)	0.110 (0.041)	0.107 (0.040)	0.123 (0.052)
Verify (for server) [s]	0.191 (0.094)	0.128 (0.059)	0.105 (0.039)	0.101 (0.038)	0.121 (0.051)

The measurements are means with standard deviations in parentheses. Communication sizes were measured using the BRAKEM$^{\text{DDL}}_{\star}$ instantiation. Daily computation times are estimated using the BRAKEM cost models. "Encapsulate / performer" is the mean time to perform all BRAKEM encapsulations for a member on a day where they perform a commit; the cost is zero on other days. "Max Decapsulate / member" is the mean time to decapsulate all BRAKEM transfers for the day, and "Verify" is the mean time to verify all BRAKEM transfers for the day. The actual mean time spent by a member to process BRAKEM transfers when applying commits would be between these values; for the server, it would be the "Verify" time.

hundreds of KiB of communication per day, tens of KiB of storage, and hundreds of milliseconds of computation per day. These costs are very low for modern devices—both the daily communication and daily computation costs are lower than the costs for a mobile device to load a typical web page [An17]. Although the daily payload size (also known as *goodput*) is an order of magnitude smaller than the communication overhead, this is a consequence of using IRC logs as the data set; the size of the payloads could be increased without bound without increasing the communication costs for the commits, signatures, or AEAD tags. In order to secure a conversation like the `#ubuntu-se` channel using Safehouse, the server would be required to store the public state, transmit almost all of the daily commits to all of the members in the session (it would not need to transmit a commit to the member that performed the operations), and it would need to spend hundreds of milliseconds per day to verify all of the commits. In practice, the bandwidth cost of distributing the commit data is likely the most significant deployment cost for the server. These results suggest that the deployment costs to secure an IRC channel like `#ubuntu-se` with Safehouse are reasonable.

The Trivial KC scheme generally performed the best for the `#ubuntu-se` data set when using $\text{BRAKEM}_{\star}^{\text{DDL}}$. This is potentially surprising: the Star scheme might be expected to outperform the Trivial scheme due to the presence of the root key making MLS.CompleteOperation (see Section 10.10.1.1) more efficient. The Trivial scheme outperformed the Star scheme when using $\text{BRAKEM}_{\star}^{\text{DDL}}$ because as shown in the performance models (see Table 10.3), the per-ring cost for $\text{BRAKEM}_{\star}^{\text{DDL}}$ is much larger than the per-recipient cost. Consequently, a $\text{BRAKEM}_{\star}^{\text{DDL}}$ transfer to approximately 64 recipients in one ring is more efficient to verify than a $\text{BRAKEM}_{\star}^{\text{DDL}}$ transfer to two rings with one recipient in each. Consider the BRAKEM transfers that occur within the two schemes when performing KC.MergeJoin (see Section 10.11.1.5 for the Trivial scheme and Section 10.11.2.5 for the Star scheme) within MLS.MassJoin (see Section 10.10.4) to merge m new members into an existing session with n members:

- **Trivial**: The m new personal keys are simply added to the key graph. A BRAKEM transfer occurs with $n + m - 1$ recipients in a ring in order to establish a new group key.
- **Star**: A BRAKEM transfer occurs with two recipients in a ring in order to combine the merging root key with the existing root key, and one recipient (the new root key) in a ring in order to establish a new group key.

Although the rings in the Star scheme are of constant size, the approach used by the Trivial scheme outperforms it when $n + m - 1 < 64$. Since the `#ubuntu-se` data set has an average of 52 members (see Table 10.1), this is usually the case. Additionally, consider the BRAKEM transfers that occur within the two schemes when performing MLS.MassEvict (see Section 10.10.5) to evict m members from an existing session with $n > m$ members:

- **Trivial**: A BRAKEM transfer occurs with $n - m$ recipients in a ring in order to establish a new group key.
- **Star**: A BRAKEM transfer occurs with $n - m$ recipients in a ring in order to replace the root key, and one recipient (the new root key) in a ring in order to establish a new group key.

The need for the Star scheme to include a second ring in the BRAKEM transfer during a mass eviction incurs another per-ring $\text{BRAKEM}_{\star}^{\text{DDL}}$ cost. For sessions with a similar number of join and evict events, this additional cost roughly doubles the average number of members that must be in the group in order for Star to outperform the Trivial scheme; the size of the `#ubuntu-se` data set is much smaller than this crossover point. Note that the Star scheme outperformed the Trivial scheme when using $\text{BRAKEM}^{\text{ZK}}$. This can be explained by the per-recipient cost to verify a $\text{BRAKEM}^{\text{ZK}}$ transfer being much closer to the per-ring cost (see Table 10.3). This also explains why the Shrub and Tree (with $bf(d) = 10$) schemes improve the estimated computation times further: these schemes generally result in BRAKEM transfers with more rings containing fewer recipients in each. However, these more complex schemes also result in much more commit data for this small data set when using $\text{BRAKEM}_{\star}^{\text{DDL}}$, because each additional BRAKEM ring requires more large $\mathbb{G}_1$ elements to be transmitted (see Section 8.2.8). In summary, for small groups like `#ubuntu-se`, the Trivial scheme is a good choice when using $\text{BRAKEM}_{\star}^{\text{DDL}}$, and the Star, Shrub, or Tree schemes are a good choice when using $\text{BRAKEM}^{\text{ZK}}$ (depending on if the developer prefers to minimize communication or computation costs).

It is important to note that the Trivial KC scheme comes with two main disadvantages compared to the Star scheme that were not measured in the experiment:

1. In KC.PrepareJoin (see Section 10.11.1.5), each joining member generates its own blinding key pair. Over time, this leads to much faster growth of the unblinding ciphertext table. The costs of aggregating the blinding public keys are deferred until the members execute MLS.Authenticate (see Section 10.10.10). This disadvantage only applies when the session is configured to use long-term authentication. Since the experiment configured the sessions for ephemeral authentication, this cost was not measured.
2. The group cannot be set into anonymous invitation mode, and it is not possible to sign payloads using an "anonymous key" known to all group members. Section 10.10.3 discussed this limitation.

In order to overcome these disadvantages, a developer might choose to use the Star scheme instead of the Trivial scheme for a group of this size when using $\text{BRAKEM}_{\star}^{\text{DDL}}$ even though the experimental results suggest that the Star scheme performs slightly worse.

10.13.4.2 Medium Group: #ubuntu-cn

Table 10.5 presents the results of the experiment for the `#ubuntu-cn` data set, representing a "medium" group. The results suggest that securing an IRC channel like `#ubuntu-cn` requires a few MiB of communication per day, under 100 KiB of storage, and just over a second of computation per day. The communication cost to download all of the commits for a day is slightly larger than the cost to visit an average web page with a mobile device [An17]. Performing a few seconds of computation per day is a very minor cost for modern devices and is unlikely to impact the user experience. These costs are completely reasonable for a secure group messaging tool running on a consumer desktop processor.

It is important to note that the BRAKEM performance models in Section 10.13.2 were derived from measurements on a consumer desktop machine using the x86-64 architecture, rather than on a slower ARM processor like the ones typically found in smartphones. Nonetheless, even if a Safehouse implementation for ARM was an order of magnitude slower than the estimates in Table 10.5, securing a group like the `#ubuntu-cn` data set with Safehouse would still only require tens of seconds per day of computation. With an appropriate user interface, this could deliver a reasonable experience for interacting with a group of this size (roughly 127 members).

The Trivial and Star KC schemes perform similarly for the `#ubuntu-cn` data set. As discussed in the context of the `#ubuntu-se` data set, these two schemes behave slightly differently when performing mass join and mass evict operations. The Star scheme uses small rings of constant size for the BRAKEM transfers during mass join operations, which generally reduces the daily communication due to smaller MLS.MassJoin commits. The average size of the `#ubuntu-cn` group (roughly 127 members), the balance between joins and evictions (see Table 10.1), and the ratio between the per-ring and per-recipient $\text{BRAKEM}_{\star}^{\text{DDL}}$ costs (see Table 10.3), all come together to cause the Trivial and Star schemes to have nearly identical computational costs when using $\text{BRAKEM}_{\star}^{\text{DDL}}$. An average of 127 members is approximately the crossover point: the Trivial scheme gains an advantage in smaller groups, and the Star scheme gains an advantage in larger groups. As with the `#ubuntu-se` data set, the `#ubuntu-cn` data set does not have enough members for the more complex KC schemes to be helpful. The Shrub and Tree schemes both add extra layers in the key graph that induce additional rings in the BRAKEM transfers, and the per-ring costs outweigh the benefits of having fewer recipients in the rings. However, the performance models suggest that the Tree scheme with $bf(d) = 10$ would achieve the lowest computation time requirements when using $\text{BRAKEM}^{\text{ZK}}$, which benefits from using more rings with fewer recipients. In summary, for medium groups like `#ubuntu-cn`, the Trivial and Star schemes are good choices when using $\text{BRAKEM}_{\star}^{\text{DDL}}$, and the Shrub and Tree schemes are good choices when using $\text{BRAKEM}^{\text{ZK}}$.

Table 10.5 PERFORMANCE EVALUATION OF SAFEHOUSE FOR A MEDIUM GROUP. This table shows the experimental results for emulating the `#ubuntu-cn` data set. When using BRAKEM$^{\text{DDL}}_{\star}$, the Trivial and Star schemes generally performed the best for this data set; both exhibited comparable performance according to all measurements. When using BRAKEM$^{\text{ZK}}$, the Tree scheme with $bf(d) = 10$ provided the best performance in terms of BRAKEM decapsulation and verification times.
(Refs: 469 [a] [b] and 472)

	Trivial	Star	Shrub $n_S = 5$	Tree $bf(d) = 10$	Tree $bf(d) = 64$
Daily update data:					
Commits **[MiB]**	3.2 (1.3)	2.68 (0.92)	2.78 (0.73)	2.72 (0.78)	2.56 (0.79)
PrepareJoin messages [KiB]	11.1 (3.6)	14.5 (5.9)	23.2 (8.4)	26 (11)	22.5 (8.4)
Daily payloads (common):					
Plaintext (goodput) [KiB]	.....		87 (30)	.....	
Signatures [KiB]	.....		134 (48)	.....	
AEAD tags [KiB]	.....		34 (12)	.....	
Storage (per-second average):					
Public state [KiB]	64 (11)	65 (11)	80 (12)	77 (18)	84 (15)
Daily computation:					
Perform probability / member	0.27	0.27	0.29	0.29	0.27
BRAKEM$^{\text{DDL}}_{\star}$ total time / day:					
Encapsulate / performer [s]	0.043 (0.014)	0.030 (0.019)	0.023 (0.009)	0.021 (0.008)	0.024 (0.010)
Max Decapsulate / member [s]	1.41 (0.43)	1.43 (0.39)	1.68 (0.42)	1.70 (0.46)	1.52 (0.44)
Verify (for server) [s]	1.33 (0.40)	1.32 (0.36)	1.54 (0.38)	1.55 (0.41)	1.39 (0.40)
BRAKEM$^{\text{ZK}}$ total time / day:					
Encapsulate / performer [s]	0.42 (0.14)	0.53 (0.15)	0.67 (0.23)	0.68 (0.24)	0.61 (0.20)
Max Decapsulate / member [s]	1.41 (0.63)	0.83 (0.36)	0.50 (0.15)	0.42 (0.12)	0.56 (0.16)
Verify (for server) [s]	1.42 (0.64)	0.83 (0.36)	0.48 (0.15)	0.40 (0.11)	0.55 (0.16)

The measurements are means with standard deviations in parentheses. The table structure and semantics are the same as in Table 10.4, except that the mean daily commit size is reported in MiB instead of KiB.

10.13.4.3 Large Group: #ubuntu

Table 10.6 presents the results of the experiment for the `#ubuntu` data set, representing a "large" group. The results suggest that securing an IRC channel like `#ubuntu` using the most efficient KC scheme (the Tree scheme with $bf(d) = 64$) requires tens of MiB of communication per day, under a MiB of storage, and tens of seconds of computation per day. These costs may or may not be reasonable, depending on the intended application. The BRAKEM shapes induced by all of the KC schemes are determined almost entirely by the group size (and partly by the structural gaps left by evicted members), so the daily computation costs are spread across all operations; the daily computation averages do not hide any extremely expensive individual operations. The session in this data set contains an average of 1 386 members (see Section 10.13.1), which is consistent with a large technical support channel or a departmental channel for a large organization. As mentioned in Section 10.13.1, the `#ubuntu` data set represents a particularly challenging scenario for a group of this size, because users join and leave the group at a relatively rapid pace: approximately 500 users join and 500 users leave every day according to Table 10.1, which is over $1/3$ of the average group size. When using MLS.Cleanup, the communication and computation costs to use Safehouse are primarily based on the quantity of membership changes (triggering mass join and mass evict events) rather than group size. Consequently, applications with more stable group membership (such as corporate communication channels) would experience lower costs. In any case, the most concerning cost shown in Table 10.6 is likely to be the daily commit size (a mean of 31 MiB for the best KC scheme), since the server must transmit this data to all members in the group. On average, this would amount to a mean of 43 GiB of traffic per day being uploaded by the server for a group of this size (with each client downloading 31 MiB).[37] For comparison, this upload cost is equivalent to a single upload of approximately 8 hours of high definition video.[38] This bandwidth cost could be reduced by using multicast networking or by switching to $\text{BRAKEM}^{\text{ZK}}$ instead of $\text{BRAKEM}^{\text{DDL}}_{\star}$.

For large groups like the `#ubuntu` data set, the Tree KC scheme clearly outperforms the others for both $\text{BRAKEM}^{\text{DDL}}_{\star}$ and $\text{BRAKEM}^{\text{ZK}}$. This is an intuitive result, because the Trivial, Star, and Shrub schemes all routinely require a BRAKEM transfer with a number of recipients that scales linearly with the group size. For a session with n existing members:

- The Trivial scheme requires a BRAKEM transfer to all n personal keys (and the personal keys for the joining members) during a mass join, and to almost n personal keys (excluding those belonging to evicted members) during a mass evict.

[37] ^ In practice, the communication sizes would be slightly smaller because the server does not need to echo a commit back to the member that performed it.

[38] ^ At the time of this writing, the YouTube video service recommended a video bitrate of 12 Mbps for 1080p60 videos encoded with the H.264 codec in the BT.709 color space.

Table 10.6 PERFORMANCE EVALUATION OF SAFEHOUSE FOR A LARGE GROUP. This table shows the experimental results for emulating the `#ubuntu` data set. The Tree scheme with $bf(d) = 64$ and BRAKEM$^{\text{DDL}}_{\star}$ significantly outperformed the other configurations for this data set in terms of commit size (the primary communication cost) and computation time. The Trivial, Star, and Shrub schemes all result in BRAKEM transfers that scale linearly with the group size, whereas the Tree scheme scales logarithmically (as the key graph grows deeper to accommodate all of the u-nodes). A branching factor of 10 instead of 64 slightly improved the computation time when using BRAKEM$^{\text{ZK}}$ because the per-recipient cost is much closer to the per-ring cost than it is for BRAKEM$^{\text{DDL}}_{\star}$. (Refs: 471 [a] [b])

	Trivial	Star	Shrub $n_S = 5$	Tree $bf(d) = 10$	Tree $bf(d) = 64$
Daily update data:					
Commits **[MiB]**	204 (6)	111 (5)	49 (2)	41 (2)	31 (1)
PrepareJoin messages [KiB]	210 (10)	460 (30)	630 (40)	710 (40)	540 (30)
Daily payloads (common):					
Plaintext (goodput) [KiB]	.....		675 (56)	.....	
Signatures [KiB]	.....		689 (56)	.....	
AEAD tags [KiB]	.....		172 (14)	.....	
Storage (per-second average):					
Public state [KiB]	545 (8)	546 (8)	657 (8)	660 (20)	570 (20)
Daily computation:					
Perform probability / member	0.001	0.001	0.001	0.005	0.005
BRAKEM$^{\text{DDL}}_{\star}$ total time / day:					
Encapsulate / performer [s]	0.37 (0.09)	0.2 (0.2)	0.06 (0.04)	0.04 (0.02)	0.03 (0.02)
Max Decapsulate / member [s]	40 (1)	27.0 (0.9)	20.0 (0.8)	21 (1)	16.0 (0.5)
Verify (for server) [s]	38 (1)	24.7 (0.8)	18.0 (0.8)	18.4 (0.9)	15.0 (0.5)
BRAKEM$^{\text{ZK}}$ total time / day:					
Encapsulate / performer [s]	1.1 (0.3)	0.9 (0.4)	0.9 (0.4)	1.1 (0.5)	0.7 (0.3)
Max Decapsulate / member [s]	111 (3)	57 (2)	14.9 (0.5)	6.2 (0.3)	7.3 (0.3)
Verify (for server) [s]	112 (3)	57 (2)	14.8 (0.5)	5.9 (0.3)	7.2 (0.3)

The measurements are means with standard deviations in parentheses. The table structure and semantics are the same as in Table 10.5. Note that the mean daily commit size is reported in MiB instead of KiB. The data for the Trivial, Star, and Shrub schemes was collected over one week of emulated time. The data for the two Tree scheme configurations was collected over one month of emulated time.

- The Star scheme requires a BRAKEM transfer to almost n personal keys during a mass evict.
- The Shrub scheme requires a BRAKEM transfer to approximately n/n_S set keys during a mass evict.

In contrast, the Tree scheme with a constant maximum branching factor $bf(d) = c$ usually requires a BRAKEM transfer with approximately $\log_c(n)$ rings (with at most c recipients in each) during a mass join or mass evict event, increasing with the number of joining or evicted members (which increases the number of paths in the grafting core). This allows the Tree scheme to scale much more efficiently for large groups. The best choice of branching factor is based on the per-ring and per-recipient costs for the BRAKEM construction: $bf(d) = 10$ performed better for $\text{BRAKEM}^{\text{ZK}}$, and $bf(d) = 64$ performed better for $\text{BRAKEM}^{\text{DDL}}_{\star}$. In any case, the Tree scheme is the best of the suggested KC schemes for large groups.

10.13.4.4 Post-Compromise Security

Table 10.7 presents the experimental results for using MLS.PCSRatchet (see Section 10.10.7) instead of MLS.Cleanup (see Section 10.10.8) in the emulated `#ubuntu-se` session with the Tree KC scheme with $bf(d) = 10$.[39] The fundamental difference between these operations is that MLS.Cleanup replaces the minimum number of keys required to eliminate superfluous edges, whereas MLS.PCSRatchet replaces all keys known to the performer. In this sense, MLS.Cleanup performs a subset of the work performed by MLS.PCSRatchet. This means that the choice has no effect on the average size of the public state (since the only structural change caused by both operations is the elimination of the same superfluous edges) or on the amount of data exchanged during KC.PrepareJoin executions (since both operations are only performed by existing members). Additionally, using MLS.PCSRatchet instead of MLS.Cleanup causes members to perform commits on a greater proportion of days, as shown in Table 10.7: MLS.Cleanup does not produce a commit on days in which the member cannot eliminate any superfluous edges, whereas MLS.PCSRatchet proceeds regardless. When an existing member performs an

[39] The Tree KC scheme with $bf(d) = 10$ was chosen for the experiment because it produces more work for MLS.PCSRatchet than the Trivial or Star schemes: the Tree scheme places u-nodes further away from the root node and therefore causes MLS.PCSRatchet to replace more keys. This effect is enhanced for smaller maximum branching factors, so the Tree KC scheme with $bf(d) = 10$ produces more work for MLS.PCSRatchet than with $bf(d) = 64$. The Shrub scheme requires slightly more work in this scenario because every u-node's keyset always includes three entries (the personal key, the set key, and the root key), whereas the Tree scheme with $bf(d) = 10$ for a session with 52 members adds $\log_{10}(52) \approx 1.7$ layers of intermediate k-notes; most u-nodes' keysets would include three entries when using the chosen Tree scheme, but some would include only two. The Tree scheme was selected instead of the Shrub scheme because it is more likely to be used in practice.

Table 10.7 POST-COMPROMISE SECURITY OVERHEAD FOR A SMALL SAFEHOUSE GROUP. This table compares the experimental results for emulating the `#ubuntu-se` data set when using MLS.Cleanup versus when using MLS.PCSRatchet. The Tree KC scheme with $bf(d) = 10$ is used in both cases. Enabling post-compromise security does not affect the communication costs of KC.PrepareJoin or the average size of the public state, but it generates more commits and correspondingly requires additional computation time. In general, the costs for the two configurations are comparable for this data set. (Refs: 473 [a] [b] and 475)

	Tree $bf(d) = 10$ MLS.Cleanup	Tree $bf(d) = 10$ MLS.PCSRatchet
Daily update data:		
Commits [KiB]	730 (270)	920 (320)
PrepareJoin messages [KiB]	5.6 (2.8)	5.6 (2.8)
Storage (per-second average):		
Public state [KiB]	40.4 (4.4)	40.4 (4.4)
Daily computation:		
Perform probability / member	0.22	0.25
BRAKEM$^{\mathrm{DDL}}_{\star}$ total time / day:		
Encapsulate / performer [s]	0.019 (0.006)	0.022 (0.010)
Max Decapsulate / member [s]	0.46 (0.17)	0.56 (0.19)
Verify (for server) [s]	0.42 (0.16)	0.51 (0.18)
BRAKEM$^{\mathrm{ZK}}$ total time / day:		
Encapsulate / performer [s]	0.63 (0.20)	0.70 (0.32)
Max Decapsulate / member [s]	0.107 (0.040)	0.137 (0.049)
Verify (for server) [s]	0.101 (0.038)	0.130 (0.046)

The measurements are means with standard deviations in parentheses. The table structure and semantics are the same as in Table 10.4. The results for the configuration using MLS.Cleanup are identical to the corresponding column in Table 10.4.

MLS.PCSRatchet operation, they are effectively performing a simplified KC.MergeJoin operation in which they are the only joining member. Based on the steady-state behavior suggested in Section 10.12, the cost of enabling MLS.PCSRatchet is less than the cost of having all existing members rejoin the group once for every forward secrecy period. In the case of the `#ubuntu-se` data set with a forward secrecy period of one week (the setting for the session emulator), this would be equivalent to having $52/7 \approx 7.4$ additional joining members per day, which is more than the actual rate (see Table 10.1). However, this is an upper bound: in practice, members that join and leave the session faster than a single forward secrecy period will not execute MLS.PCSRatchet. This type of short-lived participation is typical for the `#ubuntu-se` data set, because users often leave the channel immediately after receiving an answer to their Ubuntu technical support question. Consequently, the results shown in Table 10.7 indicate that the cost of post-compromise security is relatively low: hundreds of KiB of additional commit data per day, and approximately a hundred milliseconds (in the case of $\text{BRAKEM}_{\star}^{\text{DDL}}$) or tens of milliseconds (in the case of $\text{BRAKEM}^{\text{ZK}}$) of additional computation per day. The additional costs compared to MLS.Cleanup would increase linearly with the number of existing group members that remain in the group for longer than the forward secrecy period. A broader evaluation of the cost of using MLS.PCSRatchet is left to future work.

10.13.5 Summary of Findings

The findings discussed in Section 10.13.4 can be summarized as follows:

- When implementing Safehouse with $\text{BRAKEM}_{\star}^{\text{DDL}}$:
  - For groups with fewer than 128 members, use the Trivial scheme if anonymous invitations are not enabled and the AuthAny function is never used. Use the Star scheme if these functions are needed.
  - For larger groups, use the Tree scheme with $bf(d) = 64$.
- When implementing Safehouse with $\text{BRAKEM}^{\text{ZK}}$:
  - For groups with fewer than 128 members:
    - If computation time is the primary concern, then use the Tree scheme with $bf(d) = 10$.
    - If communication size is the primary concern, then use the Trivial scheme if anonymous invitations are not enabled and the AuthAny function is never used. Use the Star scheme if these functions are needed.
  - For larger groups, use the Tree scheme with $bf(d) = 10$.

- Enabling MLS.PCSRatchet adds additional communication and computation costs that scale linearly with the average number of members in the session. Use MLS.PCSRatchet if the security benefits of post-compromise security justify this additional cost for the intended application.

10.14 Chapter Summary

This chapter introduced Safehouse, a protocol that can be used to implement modern secure group messaging protocols and tools. Safehouse solves the central difficult cryptographic problems in the conversation security design layer (as defined in Section 2.6.2), allowing a developer to focus on creating secure group messaging tools for a particular application. Safehouse provides strong security properties and can be configured to achieve the security objectives of a wide range of deployment scenarios, as described in Chapter 7. Safehouse operates with a semi-trusted central server that forwards *commit* messages produced by members. These messages alter the state of the group in order to achieve the desired functionality (e.g., non-interactivity, dynamic membership, and secure property storage) and security properties (e.g., forward secrecy, post-compromise security, insider security, and strong deniability). The server is trusted to store an authoritative copy of the group state and to store and forward commits between group members. If the server misbehaves, the worst attack that it can perform is a denial-of-service attack that "forks" the group, partitioning its members into separate groups that can no longer communicate. This chapter introduced the specific algorithms needed to implement Safehouse, including a variety of NIZKPKs, and how these algorithms should be used. The chapter also presented the results of a performance evaluation in which Safehouse was used to secure partially synthetic IRC transcripts, providing insight into the expected costs for using the protocol.

While this chapter described Safehouse as an abstract protocol (as defined in Section 2.2), it is helpful to consider how Safehouse can be deployed in a concrete scenario. The next chapter describes a complete implementation of Safehouse that was prepared as part of this work, as well as a prototype implementation of a graphical secure group messaging tool that uses Safehouse to achieve specific deployment objectives.

CHAPTER 11 | An Example of a Safehouse Deployment

In this chapter: Implementation Discussion

When designing an abstract cryptographic protocol intended for real-world use, developing a prototype implementation of the protocol provides several benefits. Section 8.3 discussed several of these benefits in the context of implementing BRAKEM$^{\text{DDL}}_{\star}$. In particular, beyond merely demonstrating the practicality of a protocol, an implementation may uncover weaknesses or errors in the design that necessitate iteration. For example, the benefit of introducing burner keys in the BRAKEM$^{\text{DDL}}_{\star}$ construction in order to enable fixed-base exponentiation in Π_0 is an important optimization that is difficult to foresee when designing an abstract protocol. This chapter describes a prototype implementation of the Safehouse protocol and a complete group secure messaging tool developed using the protocol implementation. All of the code described in this chapter was written in the Go programming language. The prototype implementation will be made available at safehouse.im. Section 11.1 describes a high-level prototype library that allows developers to use Safehouse in most settings without concerning themselves with the details in Chapter 10, Section 11.2 describes a specific group secure messaging tool created using the aforementioned library, and Section 11.3 discusses how Safehouse could be used to develop tools for other scenarios.

11.1 A High-Level Safehouse Library

Chapter 10 described the Safehouse protocol at a low level of abstraction. A Safehouse library must implement the interface functions described in Section 10.9 and the MLS-layer operations described in Section 10.10. Using these functions and operations, a developer can implement a secure group messaging tool for any of the target scenarios introduced in Chapter 7, among others. The prototype implementation that was produced as part of this work includes implementations of all of these functions and operations using BRAKEM$^{\text{DDL}}_{\star}$ with all of the optimizations discussed in Section 8.3.

However, in most deployment scenarios, simplifying assumptions can be made that allow many of the details of the Safehouse protocol to be abstracted away, simplifying the experience for the developer. The prototype implementation exposes a high-level library that makes some of these simplifying assumptions about the intended use. The library is carefully designed to make it difficult to misuse the Safehouse protocol in an insecure manner.

The high-level library provides a `Client` struct[1] and a `Server` struct that represent agents in a Safehouse session. Each session controlled by the high-level library is orchestrated by a single `Server` struct. In other words, the library makes the simplifying assumption that the "server" agent for a session is truly a process running on a single machine. Secure group messaging protocols that require a federation of machines to act as the logical "server" must use the low-level Safehouse interface described in Chapter 10.

The high-level library does not impose a total ordering on payloads. In other words, the library does not produce a *global transcript* unless this is desirable. The developer can use the high-level Safehouse library to implement a looser notion of causality preservation, such as the notions used by the schemes discussed in Section 2.8.2.

The high-level library makes some assumptions about the configuration of the Safehouse session. The library always enables post-compromise security using the MLS.PCSRatchet operation (see Sections 10.10.7 and 10.12). By default, it sets the forward secrecy period to two weeks; members are automatically evicted and delete their local state if they are disconnected from the server for two weeks or longer. The developer can configure a different forward secrecy period using the library.

The high-level library provides functions to create and receive payload messages. Payloads are always encrypted using the XChaCha20-Poly1305 [Arc18] AEAD with a key derived from the group key $SS.K$ and the sender's agent ID. This key is produced by the SharedKey function described in Section 10.9.1 with the sender's agent ID unambiguously provided as part of the domain separation string. Each member selects AEAD nonces for its outgoing payloads sequentially.[2] The nonce sequence is reset when a new group key (and therefore a new key for the AEAD) is established. Payloads are also signed using the sender's personal key, $\text{personal}(U_{\text{perf}})$, and the server verifies the signatures before relaying the payloads to other members. For performance reasons, the developer can configure the library so that the server refrains from verifying the signatures. The library also allows a "public tag" to be attached to payloads. The public tag is an

[1] ^ A "struct" in Go is similar to a "class" in object-oriented languages: a Go struct is a collection of named data fields, and functions can also be attached to structs.

[2] ^ This approach is safe for the chosen AEAD, and it ensures that the nonces can be highly compressed when attached to the ciphertexts. Although different members may use the same nonces, this is acceptable because each member uses a unique AEAD key derived from the group key. Individual members never reuse nonces with the same key.

opaque binary blob that is authenticated but sent as plaintext—it is authenticated by providing it to the AEAD as "associated data". The state hash $PS.H$ is also always included in the "associated data", even for payloads with no public tag. This ensures that messages can only be transmitted between members sharing the same public state, as recommended in Section 10.1.1.

All cryptographic hashes in the prototype are implemented using cSHAKE128 [KCP16]. cSHAKE128 is an XOF derived from the SHA-3 sponge construction. Importantly, it provides a standardized domain separation mechanism that ensures that hashes for different contexts do not collide, even for the same preimages. Moreover, as an XOF, cSHAKE128 provides an arbitrarily long output value. This is convenient for implementing rejection sampling when generating uniformly random integers in a given range, which is a common requirement in Safehouse. The implementation of the SharedKey function (see Section 10.9.1) uses cSHAKE128 to derive keys from the group key. The SharedKey implementation unambiguously provides both the developer's domain separation string Φ and a constant string specific to the Safehouse implementation as inputs to cSHAKE128's domain separation mechanism. The SharedKey implementation also includes a unique encoding of the group key, $SS.K$, and the requested key length, ℓ as the cSHAKE128 preimage. This ensures that the cSHAKE128 output is not a prefix of outputs for longer key lengths.

By default, the high-level library limits group sizes to 2,000 members, payload sizes to 1,200 bytes, and public tags to 50 bytes. These limits can be reconfigured by the developer. The default group policy (see Section 10.1.6) permits payload transmissions, mass join operations, mass evict operations, invitation issuances and retractions, and modifications of the confidential label-value store by all members. The developer can also provide an implementation of a different group policy.

When a session is created using the high-level library, the client specifies the deniability and authentication ephemerality modes (see Section 10.1.6). If the developer wishes to limit the protocol to specific modes, a custom function can be provided that is executed by the server to verify the initial public state for the session. The library automatically stores the deniability and authentication ephemerality configuration in the developer mode of the group state, $PS.M$. If the developer provides an opaque mode string, it is transparently appended to these internal options. The developer can specify whether invitations should be anonymously issued or not (see Section 10.1.6) as part of the library configuration.

The high-level library begins communications with an internal protocol version number. This allows for future expansion. Unlike the cipher negotiation mechanism historically used by TLS,

the cryptographic primitives used by the library are implicitly defined by the protocol version. The library always implements Safehouse using $\text{BRAKEM}_{\star}^{\text{DDL}}$.[3]

The aforementioned configuration options are hashed to produce an *options fingerprint*. When a client creates a new session or joins an existing session, it first compares options fingerprints with the server to ensure that both endpoints agree on the configuration of the high-level library. This is a convenient "fast fail" mechanism and not a security requirement—if this check were omitted, then the client would eventually reject commits or payloads that are allowed by the server's configuration, but not by the client's configuration. This would eventually lead to the client being evicted from the group due to diverging from the server's public state and being unable to perform MLS.PCSRatchet commits.

11.1.1 Server Design

To set up a server with the high-level library, the developer first initializes a `Server` struct in the program. The developer is responsible for accepting network connections (represented as `net.Conn` interfaces from the Go standard library) from members. The developer then calls the `Handle` function on the `Server` struct with the connection. The library takes complete control of the connection. By default, the library checks to ensure that network connections to the server were established using TLS (but without requiring client authentication, which would undermine the deniability properties of the scheme). When calling `Handle`, the developer can specify configuration options: the expected agent ID of the connecting client, a function that validates the public state of a newly created session, a *message router* to use, a *storage system* to use, and whether to allow non-TLS connections to the server. The message router and storage system mechanisms are discussed later in this section.

The `Server` struct provided by the library exposes aspects of the public state to the developer. The following values are exposed:

- The labels and values stored in the public label-value store, $PS.A$.
- The labels (but not the values) stored in the confidential label-value store, $PS.E$.
- The agent IDs of the members in the session, which are bound to u-nodes in the key graph, $PS.G$.

[3] The prototype includes an incomplete implementation of $\text{BRAKEM}^{\text{ZK}}$ that is developed enough to produce the performance evaluation in Section 8.6, but not enough to implement the whole Safehouse protocol. Completing the $\text{BRAKEM}^{\text{ZK}}$ implementation in the prototype is left as future work.

- The identifiers, types, expiry dates, and remaining uses of layaways in the layaway table, *PS.L*.
- The deniability mode, authentication ephemerality mode, and developer mode, all stored in *PS.M*.

The other values stored in the public state, as discussed in Section 10.1.2, are low-level details that are irrelevant in most applications. In order to prevent the possibility of misuse, the high-level library does not expose these other values to the developer.

Once a connection is given to the `Server` struct for handling, an initial protocol handshake takes place. The client specifies whether it is creating a new session, joining a session, or reconnecting as an existing member. The server rejects the connection in invalid situations (e.g., if a client asks to create a new session but the server's group state is already non-empty).

The developer may optionally supply a message router when instantiating the server. The message router is responsible for scheduling the delivery of payloads and commits, as well as replaying old payloads and commits when a client reconnects to the server. Newly received payloads and commits are provided to the message router for delivery scheduling. This powerful abstraction allows the developer to implement an arbitrary concurrency policy that yields the desired causality preservation property. In order to support the store-and-forward functionality of the message router, the high-level library assigns monotonically increasing numbers to group states in the session. Numbers are also assigned to payloads, but this assignment is controlled by the message router. The message router is responsible for assigning a number to payloads when they are routed. When clients reconnect to the server, they provide the numbers of the most recently received group states and payloads; this allows the message router to schedule delivery of stored commits and payloads that were processed since that time. Message router implementations must be able to replay the linear sequence of commits (at least within the forward secrecy window), but payloads do not necessarily need to be replayable—for high-throughput real-time applications like multimedia transmission, the developer might supply a message router that does not store payloads. The message router can also selectively deliver messages to a subset of members (e.g., based on the public tag attached to the payload) if desired.

The developer may also supply a storage system when instantiating the server. The storage system is responsible for storing the server's current group state, *PS*, which is treated as an opaque blob of data. By default, the high-level library does not store the server's state; developers can trivially provide custom implementations that store the data in a filesystem or database.

The server implementation in the prototype high-level library makes extensive use of concurrency and synchronization mechanisms. Alternative implementations of the Safehouse protocol will likely need to use similar locking techniques. The server's group state is protected by a

read/write lock. The semantics of this lock, defined by the Go standard library, are that acquisition of the read lock blocks while an acquisition of the write lock is pending. There are two queues for requests to acquire the write lock: a high-priority queue for members wishing to perform ordinary commits, and a low-priority queue for new members that wish to join the session. Commits from existing members are always serviced before mass join operations; this prevents a denial-of-service attack in which outsiders continuously request to join the group, and also ensures that the mass join mechanism is used more efficiently.

Members attach the public state number when sending a payload. When a payload is received from a member, the read lock is acquired. If the server notices that a payload was sent for an old public state, it sends a rejection notification to the client and discards the payload. Otherwise, the payload is provided to the message router for delivery scheduling. Once the message router schedules the payload for delivery, the server sends an acknowledgment to the client.

When a member requests to perform a commit, the server adds the request to the back of the high-priority queue. When the server removes a request from the high-priority queue, it informs the associated client that it may perform its commit. The client must then transmit the commit within a configurable time limit. Upon receipt, the server acquires the write lock for the group state, applies the commit, stores the updated group state, increments the group state number, and then provides the new group state to the message router for delivery. The server then releases the write lock and moves on to the next request, if one is pending.

It is important to note that this locking design allows the server to continue to successfully route payloads at almost all times. Payloads, commits, and handshakes are generally handled concurrently. In particular, payloads can continue to be delivered while a member is authorized to perform a commit, but before the commit is received. This is an important feature because some devices may require a noticeable amount of time to perform a commit, especially for low-power devices in sessions with many members. The only time in which payloads are rejected is when the server is in the process of applying a commit, because the payloads will be encrypted using keys derived from an outdated group key.

The high-level library provides a default implementation of a message router that is suitable for most secure group messaging applications. This default message router induces a total ordering of payloads, thereby providing a global transcript. Payloads are assigned monotonically increasing numbers. The commits and payloads are stored in a configurable storage location in sequential order. Old commits and payloads are retrieved from storage in order to replay transmissions for reconnecting clients. The message router has a configurable *maximum backpressure* that specifies how many commits and payloads may be pending for delivery to a client. If the maximum backpressure is exceeded, the message router instructs the server to drop the client connection. This protects the server against denial-of-service attacks in which a client stops processing

transmissions. By default, the high-level library stores commits and payloads for the default message router in memory; the developer can provide a custom implementation to store the data in a non-volatile location, such as a database.

A *joining manager* routine concurrently accepts requests from new members to join the session, alongside the invitation identifiers that the joining members intend to use when joining. The joining members also provide a domain-separated cryptographic hash of the invitation public key in order to ensure that the invitation identifier refers to the expected invitation key pair (see Section 10.10.3 for a discussion of identifier reuse). When at least one new member is waiting to join, the joining manager submits a request to perform a commit. This request is added to the low-priority queue. When the server has processed all commit requests in the high-priority queue, it grants permission for the joining manager to produce a commit. The server can be configured so that the joining manager delays starting the mass join procedure for a minimum amount of time (one second by default) in order to aggregate sudden bursts of join requests. When the joining manager is granted permission to perform a commit, it sends the public state to each of the joining members. For each joining member, the public state is encrypted to the invitation public key corresponding to the invitation identifier for that member; this implicitly prevents the contents of the public state from being leaked. The joining manager selects a representative for the joining members at random, then distributes the set of agent IDs and the representative's agent ID. The joining members then perform the KC.PrepareJoin protocol introduced in Section 10.6. During the KC.PrepareJoin procedure, the server routes transmissions between the joining members by receiving messages and recipient lists over the corresponding `net.Conn` interfaces; for most KC schemes, these are transmissions for the authenticated TKLL protocol. Finally, the joining manager receives a commit with the MLS.MassJoin operation (see Section 10.10.4) from the representative and verifies that the commit adds the expected joining members to the session. The server then acquires the write lock for the group states, applies the commit, and distributes the commit as with operations performed by existing members. The server can then resume its processing of commit requests.

11.1.2 Client Design

Group members can be implemented using the high-level library by creating a `Client` struct. When creating the `Client`, the developer provides an established net.Conn connection to the server (typically an established TLS tunnel) along with connection options. Among these options, the developer must specify whether to initialize a new session, join an existing session honestly, join an existing session by forging authentication (for deniability purposes), or reconnect to a session. Reconnecting to a session requires access to previously stored state. Joining an existing session requires the appropriate private keys, as discussed in Section 10.8: to join honestly, the

developer must provide the user's identity private key (if long-term authentication is enabled for the session) and the invitation private key; to join using the deniability mechanism when the session is not in "non-repudiable" mode, the developer must provide either the personal private key or the identity private key of every member in the session.[4] The library provides functions to generate a new identity key pair, which is exposed as an opaque binary blob.

Once created, the developer can interact with the `Client` struct in four ways:

1. The next *event* that occurs in the session can be read.
2. When no event is being read, an abstraction of the group state can be inspected.
3. A payload can be sent to the group.
4. The group state can be modified.

The developer is expected to concurrently read the next event from the `Client` at all times, except when it is deemed necessary to inspect the group state. The events are descriptions of payloads that have been received or operations that have been performed; this is the mechanism that allows the secure group messaging tool to react to communication. All events include the server time that the payload or commit was first handled and the sender's agent ID (except for anonymous issuances of invitations). A special event indicates the completion of the historical event replay upon reconnection.

The developer can inspect an abstraction of the group state stored in the `Client`. This abstraction includes all of the information from the public state that is also made available to the `Server`, as discussed in Section 11.1.1. In addition, the state exposed by the `Client` also includes (decrypted) values in the confidential label-value store, (unblinded) identity public keys for the members, and whether or not members have authenticated their identities (see Section 10.8).

The library provides access to the SharedKey function (see Section 10.9.1) so that the developer can derive shared keys for use in the application. However, a constant string is prepended to the domain separation strings provided by the developer in order to prevent applications from deriving the same key that is used by the high-level library to encrypt payloads. This is a precautionary defense mechanism to prevent misuse of keys.

The high-level library allows payloads, with a given public tag, to be sent either "reliably" or "unreliably". When sent reliably, the `Client` will wait to receive a receipt acknowledgment

[4] ^ For sessions in the "non-repudiable" mode, it is not possible to join using forged authentication. Sessions in the "offline deniable" mode require the personal private keys for all members. Sessions in the "strongly deniable" mode require either the personal private key or identity private key for all members. See Section 10.8 for a detailed description of the authentication mechanism.

or rejection from the server before returning. A payload may be rejected if the server applies a new commit while the payload, encrypted using a key derived from an outdated group key, is still being transmitted; in these cases, the developer would typically retry the transmission. When sent unreliably, control immediately returns to the developer's code. Unreliable payload transmissions can occur concurrently with other unreliable or reliable transmissions.

MLS-layer operations (see Section 10.10) are not directly exposed to the developer by the high-level library. Instead, the developer can ask the `Client` to modify the group. This sends a request to the server for permission to perform a commit, which adds the member into the high-priority queue discussed in Section 11.1.1. Once the server informs the `Client` that it may perform the commit, control is transferred to a developer-supplied callback function. This function is given access to the abstraction of group state described previously, with its values in a stable state immediately prior to the modification. The function is also given access to simplified abstractions of the MLS-layer operations that can be performed. Instructions to perform operations are passed along to an underlying MLS-layer performer context. Upon completion of the callback function, the commit is produced and sent to the server. Control is returned to the developer upon acknowledgment from the server that the commit has been handled by the message router (see Section 10.8). This design takes into account the possibility that the group state stored by the server may change in the time between the developer's request to modify the group and when permission to perform an MLS-layer operation is finally granted. The developer is responsible for inspecting the group state and determining whether the desired modifications are still necessary; in some cases, commits submitted by other members may render the intended commit redundant.

The `Client` automatically handles the expected steady-state behavior discussed in Section 10.12. The `Client` automatically authenticates itself to new members that join the session using MLS.Authenticate (see Section 10.10.10), retracts invitations that have expired using MLS.RetractInvitation (see Section 10.10.3.3), regularly performs MLS.PCSRatchet operations (see Section 10.10.7), and evicts other members from the group when they have not performed a MLS.PCSRatchet operation within the forward secrecy window.

11.2 Example Application: Secure Internet Relay Chat

As part of this work, a complete secure group messaging tool was implemented using the high-level Safehouse library discussed in Section 11.1. The purpose of this example application is to demonstrate how a developer might use Safehouse to build a practical tool for one of the target scenarios described in Chapter 7. Of course, Safehouse is applicable to many scenarios—the example application merely demonstrates one possible use.

The example includes a graphical desktop client application and a server program that approximate the functionality of a modernized IRC chat server. In particular, unlike IRC, the example application's user experience supports non-interactivity: users can close and reopen their client (within the forward secrecy period) and receive all messages that they missed while offline. The graphical client allows users to log in with a username and password combination, with support for password changes. This type of authentication is very familiar to users and is more accessible than explicit management of private keys.[5] Like IRC, the example supports multiple independent *channels*, each with its own list of participants and chat history. The server stores a list of accounts and which channels each account is participating in.

Each channel in the example is implemented as an instance of a `Server` struct in the high-level library (i.e., each channel has its own independent Safehouse session). For simplicity, the example server does not support federation like IRC; all accounts and channels are operated by a single server process. From a privacy perspective, the example server is permitted to link members' activity across channels, but Safehouse ensures that the server cannot unblind the long-term identities associated with the members. The server assigns a globally unique agent ID to each account. The agent ID is reused by the client for each channel in which they are a member. Like IRC, the example allows channels to have an associated *topic*, which is a persistent text string that always appears in the chat display for the channel. This value is stored in the confidential label-value store. The example application configures the high-level Safehouse library with the default settings, including the default message router, which produces a global transcript for each channel. The Safehouse sessions are configured in the "offline deniable" mode with long-term identification.

The following sections describe the example application in greater detail. Section 11.2.1 describes the design of the server and the communication protocol, Section 11.2.2 describes how the graphical client encrypts its state, and Section 11.2.3 depicts and describes the user interface.

11.2.1 Server and Protocol Design

The example application is designed to support the classic username and password authentication model, while also making it easy to change the password. At a high level, these requirements yield a simple design: the identity key pair for use in the Safehouse protocol is randomly generated, the identity private key is encrypted using a key derived from the password, and the encrypted identity private key is stored on the server. This design allows a client with no state other than the username and password to recover the identity key pair and resume participation in active

[5] The usability studies surveyed in Section 2.1 routinely found that it is difficult for most users to create accurate mental models of public-key cryptography.

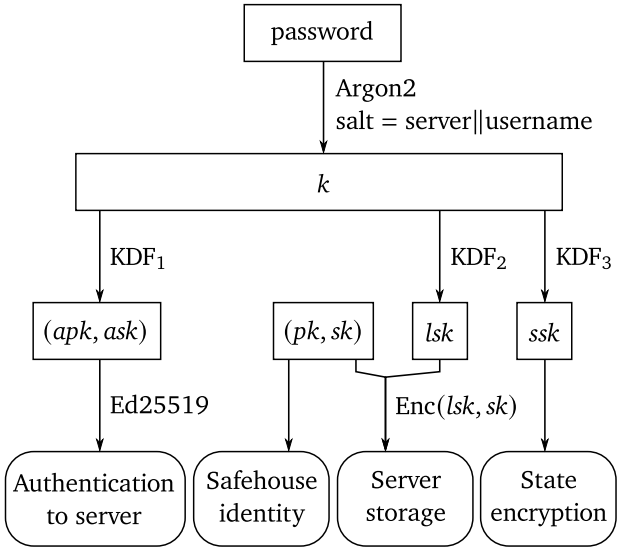


Figure 11.1 CLIENT KEY DERIVATIONS IN THE EXAMPLE APPLICATION. (pk, sk) is the long-term identity key pair used in the Safehouse sessions; it is generated independently of the password. The server name, username, and password are used to derive a key pair and two symmetric keys. (apk, ask) is an Ed25519 key pair used to initially authenticate to the server. sk is encrypted using lsk as a key and the ciphertext is stored by the server, retrievable after signing a nonce with ask. ssk is used as the master key for the system that encrypts the state stored by the client device.
(Refs: 487, 489, and 490 [a] [b])

Safehouse sessions, which is necessary for multi-device support. While the prototype does not yet support multiple devices sharing the same account, the authentication structure is designed to anticipate this as a future feature; this topic is revisited in Section 11.3.

Figure 11.1 depicts the cryptographic key material involved in implementing the aforementioned authentication scheme. The client derives a master key, k, from the user's password using a password-based key derivation function: Argon2 [BDK16]. cSHAKE128 [KCP16] is then used as a KDF to derive the *authentication key pair*, (apk, ask), the *locking key*, lsk, and the *storage master key*, ssk, from k. sk is then encrypted using XChaCha20-Poly1305 [Arc18] with lsk as the key. When registering an account on the server, apk and the encrypted version of sk are stored as part of the account information. When logging in, the client sends the username, the server responds with a challenge nonce, and then the client replies with an Ed25519 signature [BDL+12] on the nonce using the signing key ask. After verifying the signature, the server sends the encrypted form of sk to the client, which can decrypt it using lsk. The presence of the authentication key pair ensures that adversaries cannot download an encrypted identity private key in order to guess the password and recover lsk in an offline brute force attack. It is also important that the

authentication protocol occurs within a secure channel, because an adversary can perform an offline brute force attack against the password if it gains access to a valid signature produced using *ask*.[6] As discussed in Section 11.1.1, the high-level library ensures that Safehouse traffic is contained within a TLS tunnel by default; the example performs the authentication protocol within a TLS tunnel that is subsequently used for the Safehouse communications. After logging in, the server sends the client's agent ID and a list of channels that it has joined. The identity private key, *sk*, is generated independently of the password in order to support password changes: the account data is simply updated with a new authentication key pair and an encryption of *sk* using the new locking key.

As discussed in Section 11.1.1, the default message router provided by the high-level library can store payloads and commits in a configurable location. The example server stores the payloads and commits in a relational database that is accessed using SQL. This database also stores all of the accounts and channel data. Each channel is identified by a unique name. The high-level library is configured with a storage system that stores the state for each `Server` struct in the database row for the associated channel. The payloads and commits are stored as rows containing the sequence number, the data itself, and the server timestamp at the moment that the payload or commit was handled by the message router.

The connection between the client and the server begins as a TLS tunnel over TCP with server authentication and OCSP stapling, but without client authentication. The example server is capable of automatically downloading new OCSP responses to staple and automatically reloading the TLS certificate. Once the TLS handshake is complete, the `yamux` connection multiplexer[7] is used to facilitate multiple independent "streams" within the same TLS connection. These streams function similarly to the multiplexing approach used by the HTTP/2 protocol, except that the example application uses them primarily for labeling and logical organization rather than for performance.[8] The initial registration or log-in process occurs over the default stream, called the *control stream*. The client can open a new *channel stream* at any time. In a channel stream, the client first sends the name of the channel. The server instantiates a `Server` struct for the channel if one does not already exist, and then sends the `yamux` stream to the `Server` struct for handling, as discussed in Section 11.1.1. The high-level Safehouse library then handles creating, joining, or reconnecting to the session.

[6] ^ The server can always perform offline brute force attacks against the accounts (e.g., by using *apk* as a verification oracle), which is a downside of the approach used in this example.

[7] ^ `https://github.com/hashicorp/yamux`

[8] ^ Since the streams produced by `yamux` are all broken into labeled frames that are transmitted inside the same TCP connection (inside a TLS tunnel), they are subject to the same *head-of-line blocking* as HTTP/2: if a packet is dropped and must be retransmitted, all streams in the connection stall.

After successfully registering or logging in, the client can request the username associated with an agent ID by sending a query in the control stream. This allows the client to display usernames in the channels. The trust establishment mechanism, which is discussed in Section 11.2.3, ensures that the usernames and identity public keys are bound to the correct entities.

11.2.2 Client State Encryption

The graphical client for the example application stores secret state for the Safehouse sessions in the local filesystem. This data should be encrypted independently of disk or filesystem encryption mechanisms that may be present.[9] For this reason, the example includes a state encryption subsystem in the client.

At a low level, the client implements a "block device" that stores encrypted data in 4,056-byte blocks indexed by position. When combined with a 24-byte nonce and 16-byte authentication tag, this quantity is aligned with the 4 KiB block size used by most modern storage devices. Each "block device" is simply an abstraction layer on top of a simple file; this abstraction is not a "block device" in the sense used by Linux- or BSD-based operating systems. The first block in the file contains the 32-byte *block device key* padded to 4056 bytes and encrypted using XChaCha20-Poly1305 with *ssk* (see Figure 11.1) as the key and a random nonce. The nonce and authentication tag are appended to the 4056 bytes of ciphertext. Each subsequent block is encrypted using XChaCha20-Poly1305 a block-specific key derived from the block device key and block position using cSHAKE128 as a KDF. Every time that data in the block device is changed, a new nonce is generated at random and the corresponding block is entirely re-encrypted; this does not suffer a performance penalty in practice as long as the blocks are the same size as the blocks in the underlying device. If the storage master key, *ssk*, is replaced with a new storage master key, *ssk′*, because the user changes the password, then the block device key can simply be re-encrypted using *ssk′*; only the first block in the file needs to be updated.

The client stores an index file with a list of stored connection configurations. Each stored connection contains a server address, a username, the client's agent ID, the last connection time, and a channel list. The client can consult the timestamps in the index file and compare the stored channel list with the channel list received upon reconnection in order to identify channel states that should be securely erased, either because the client has not reconnected within the forward

[9] Modern operating systems are increasingly introducing security barriers between processes running under the same account. Encrypting the local state of a secure messaging client is a barrier of this type that can mitigate some information leakage attacks. Additionally, cryptographically enforcing the presence of the password in order to decrypt the client state aligns the cryptographic guarantees with the expectations established by the user interface, which requires a password to connect to a session.

secrecy period, or because the server has refused to reconnect the client to the channel. The index file is unencrypted and unauthenticated, because it must be read prior to the input of a password—the index file is a list of server connections, each of which requires a password. If the kernel allows a malicious process to edit the index file, there are two potential negative outcomes: the malicious process could erase the client's knowledge of a legitimate session (which would eventually cause the other members to evict the client due to a timeout), or it could configure a new session (potentially for a connection to a malicious server). However, since the server hostname is included in the derivation of the master key (see Figure 11.1) and Safehouse is designed to tolerate malicious servers, causing a user to connect to a malicious server does not undermine any of the security properties.[10]

The client creates two encrypted block devices for each channel in each server: a chat log, and a state file. These block devices use a storage master key derived from the server, username, and password for the connection, as depicted in Figure 11.1. The chat log stores the history of all events that have occurred in the channel in such a way that a range of events can be rapidly accessed by their sequential index. The state file contains the group state for the `Client` struct, exported using the high-level Safehouse library. The state is restored when the client starts again, allowing the Safehouse protocol to continue.

The file format of the chat log must be carefully designed in order to ensure that positional event lookups are extremely fast; this is an important requirement of the user interface implementation described in Section 11.2.3. The file contains three types of blocks: chatter blocks, index blocks, and event blocks. Each chatter block stores a list of mappings from numeric chatter IDs to usernames. These chatter IDs are local to the chat log and do not necessarily match the agent IDs that correspond to the usernames. The mappings stored in the chatter block are compressed using a lossless compression algorithm. The chatter block ends with the index of the next chatter block in the file. Index blocks occur at a regular interval in the file: every 507 blocks. The index block contains a list of the first event indices that appear in the next 506 blocks, if they are event blocks (or a special value if they are chatter blocks). These values are fixed-length integers and

[10] ^ In practice, a malicious process that can edit the index file could mount an elaborate "phishing" attack in which it configures a connection to a malicious session that tricks the user into sending messages that they would not normally send. This attack would be bolstered if the malicious process also had access to the user's password and storage files for the legitimate session, because then it could set up the malicious session to replicate the legitimate session's conversation history. These attacks could be partially mitigated by requiring an additional secret (e.g., another password or a hardware security module) to add integrity protection to the index file, but this secret would need to remain uncompromised by the adversary. The example application does not include this extra layer of protection. If the example application did not use the client state encryption described in this section, then a powerful adversary with a malicious process running on the user's device would be able to recover and exfiltrate the user's conversation histories and group states without access to the user's password, rather than being limited to these phishing attacks.

are not compressed. The periodicity of the index blocks is determined by the block size and the number of bits used to store block indices: $507 = {}^{4056}/_{8}$. Event blocks store serializations of events with all of the information necessary to draw them in the user interface, compressed using a lossless compression algorithm. Events that refer to usernames do so by using chatter IDs.

When a new event is added to the chat log, the client first attempts to append it to the latest event block, compress the new data, and write the updated plaintext to the block device. If the new data does not fit, then a new event block is appended to the file, along with a new index block if necessary. During operation, the client caches the entire mapping from chatter IDs to usernames in memory.[11] If a new username is encountered when writing to the chat log, it is added to the latest recent chatter block; in a similar manner as for event blocks, a new chatter block is added if the new mapping does not fit. This design allows for very fast binary searches of the encrypted chat log to recover all events within a given range. Additionally, the client uses a caching layer with a least-recently-used eviction policy in order to cache recently queried events.

For each server, the client also stores a trust database in a block device. The trust database stores the results of the trust establishment mechanism: a list of usernames, the associated (unblinded) identity public keys, and whether or not the user has verified that those values correspond to the expected identity.

11.2.3 GUI Design and Implementation

The graphical user interface for the example application was designed to be similar to a modern IRC client, demonstrating that it is possible to create familiar and simple user experiences using Safehouse. The interface, depicted in Figures 11.2, 11.3, 11.4, and 11.5, was implemented using the Gio library.[12] The Gio library is an *immediate mode* graphics toolkit, which is an alternative to traditional *retained mode* toolkits such as Qt or GTK. An immediate mode library operates similarly to a video game engine: the developer sends the commands to repaint the entire interface during each frame of rendering. A caching layer makes this process efficient. Aside from simplicity, the main advantage of immediate mode libraries is that they support rapid changes in large portions of the interface, since the library does not store any state information that needs to be updated when the interface is modified. This makes immediate mode libraries particularly suitable for chat applications with very long conversation histories: the chat log can be scrolled extremely efficiently, with the frame rate limited by the size of the window instead of the size of the chat log. The design of the encrypted chat log storage system discussed in Section 11.2.2, which

[11] ^ This is sufficient for the example application, in which the total number of usernames over time is expected to be small. Other applications using a similar storage design could implement a cache eviction policy if necessary.
[12] ^ https://gioui.org/

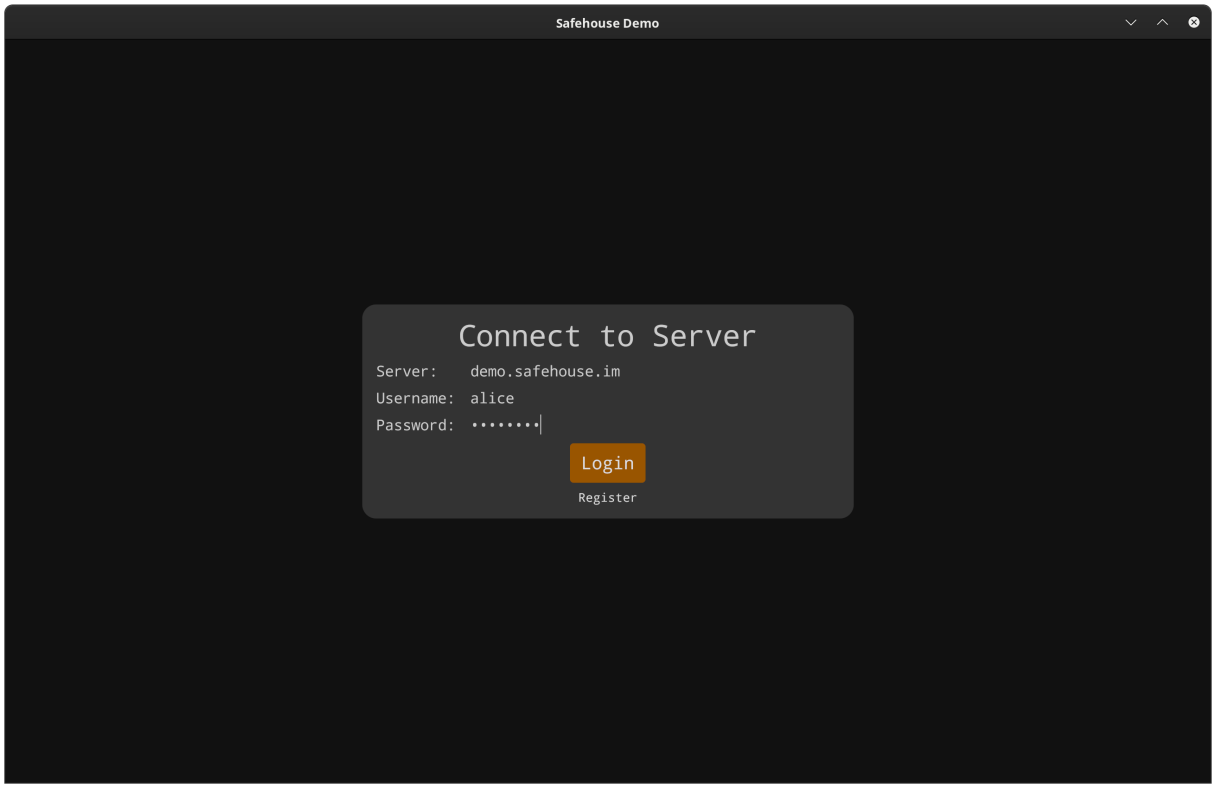


Figure 11.2 THE CONNECTION SCREEN IN THE EXAMPLE APPLICATION. Alice enters a server address, username, and password in order to derive the appropriate keys and connect to the server. (Ref: 491)

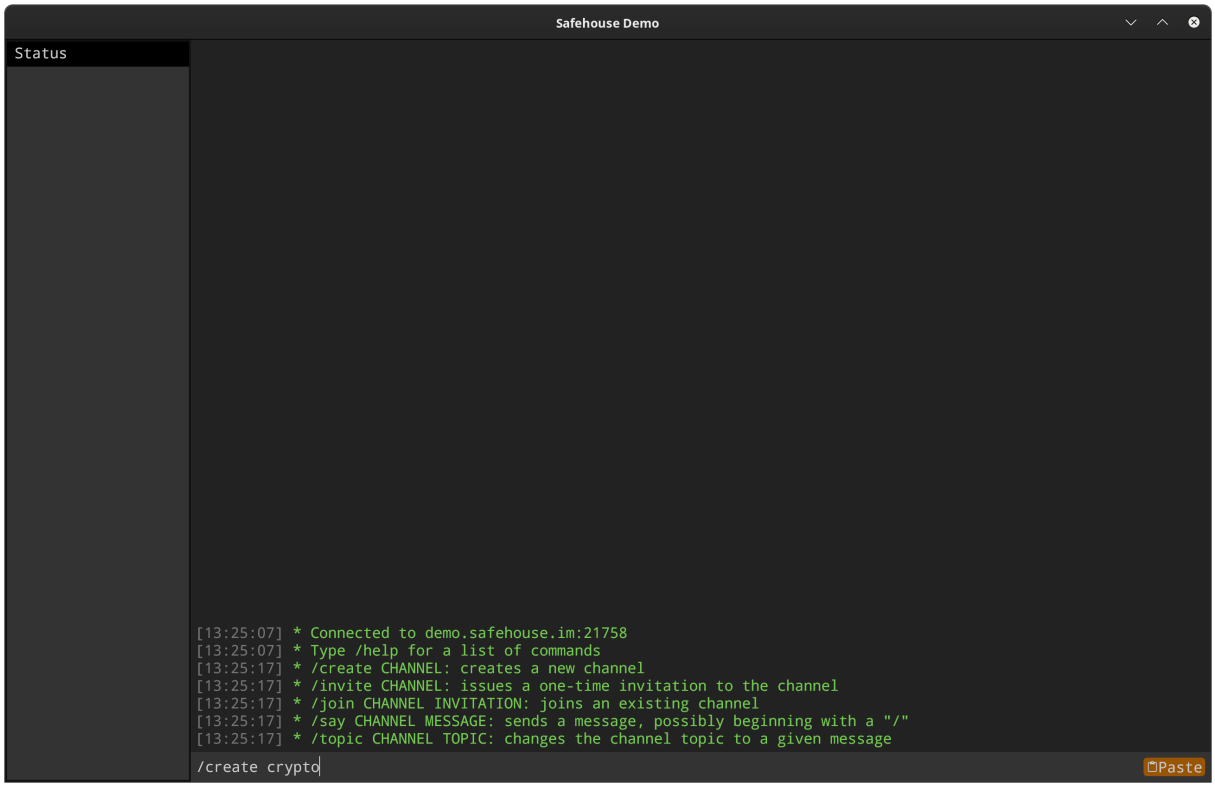


Figure 11.3 THE STATUS SCREEN IN THE EXAMPLE APPLICATION. Alice encounters this status view upon logging in. This screenshot shows Alice's view prior to joining any channels. Alice uses the `/create` command to create a new channel called `crypto`, which starts a new Safehouse session. (Ref: 491)

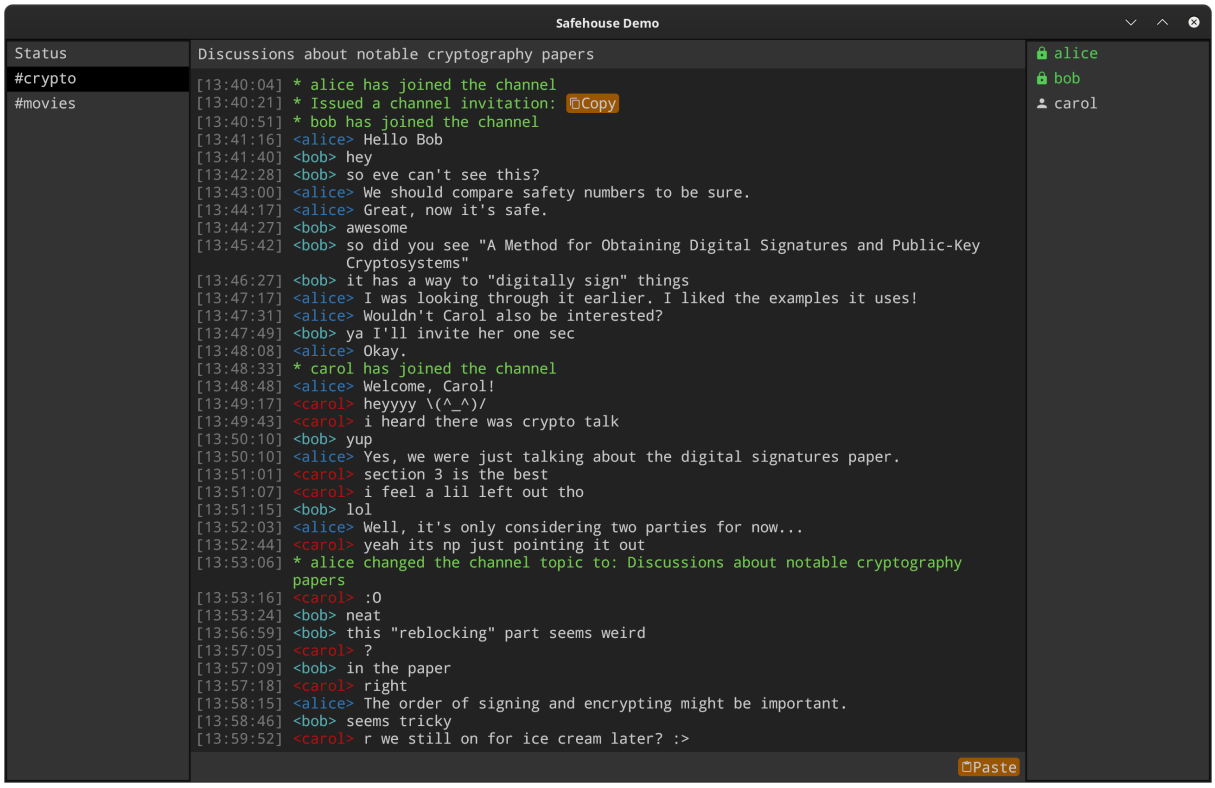


Figure 11.4 THE CONVERSATION SCREEN IN THE EXAMPLE APPLICATION. Alice securely communicates with Bob and Carol in the `crypto` channel. Alice has performed trust establishment with Bob, but not with Carol; this is indicated by the lock iconography in the user list on the right. The channel topic is securely set in the confidential label-value store and displayed at the top of the chat log. (Ref: 491)

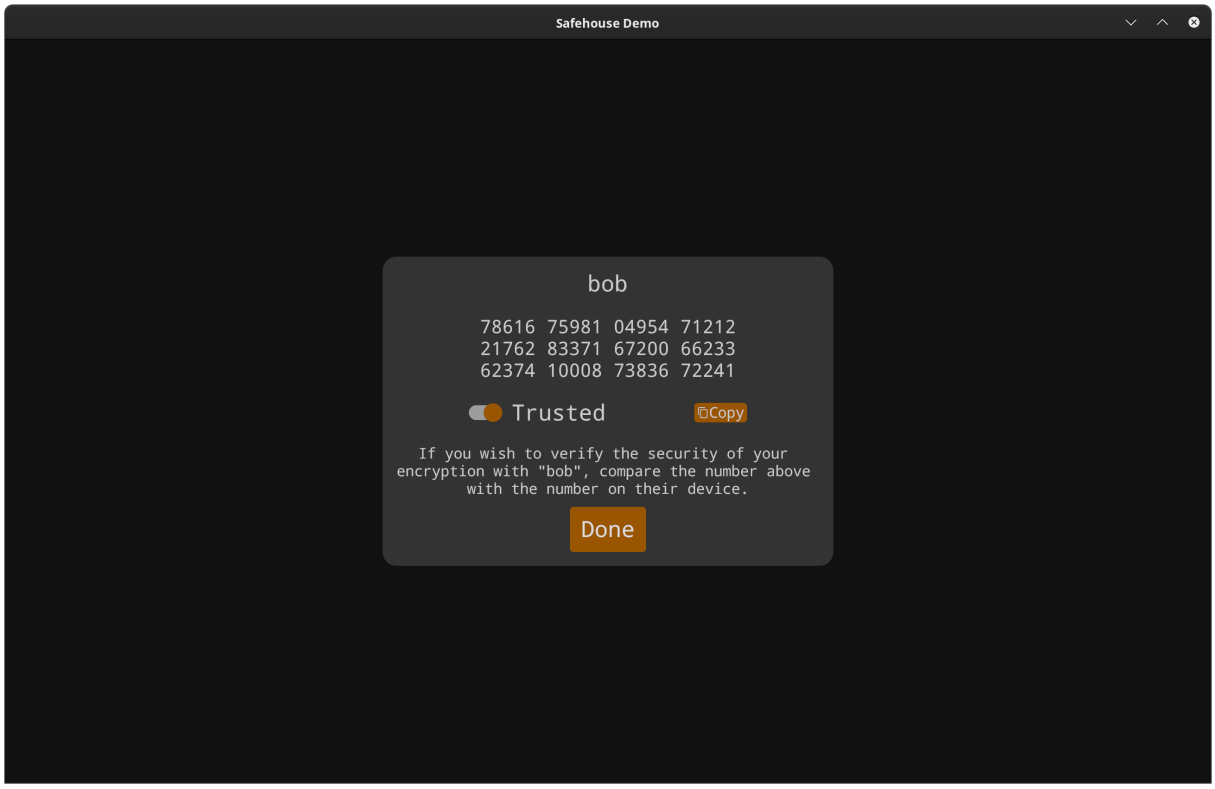


Figure 11.5 THE TRUST ESTABLISHMENT SCREEN IN THE EXAMPLE APPLICATION. This screen is shown to Alice when verifying the identity of Bob. The same set of numbers will appear to Bob when verifying the identity of Alice. (Refs: 491 and 496)

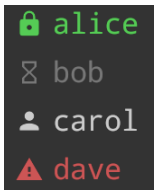


Figure 11.6 TRUST LEVEL SYMBOLOGY IN THE EXAMPLE APPLICATION. Alice's identity public key has been authenticated and trust is established. Bob's identity public key has not yet been authenticated. Carol's identity public key has been authenticated but trust establishment has not yet been performed. Dave's identity public key does not match the one stored in the trust database. (Ref: 496)

allows a range of historical events to be quickly retrieved based on their sequential position, is important for achieving good performance when using an immediate mode rendering library.

The example application borrows the "safety number" system that is used for trust establishment in Signal [Ope13]. The trust establishment process is depicted in Figure 11.5: each pair of users ensures that their "safety number" matches using an authenticated out-of-band channel. The safety numbers are generated using cSHAKE128 with rejection sampling to produce the 5-digit numbers. The hash input is a canonical ordering of the two usernames and the (unblinded) identity public keys. Once a user has verified the safety number and completes the trust establishment, the result is stored in the trust database for the server. Usernames in the user list for a channel are displayed with symbology that indicates the status of authentication and trust establishment, as depicted in Figure 11.6. Notably, the design of Safehouse means that some members may be "identified" but not yet "authenticated" (see Section 10.1.9). The client in the example application depicts these users with a gray hourglass symbol, regardless of any trust establishment that may have been performed for the claimed (but not yet authenticated) identity public key.

The example application uses IRC-like commands to issue invitations and join channels instead of graphical elements; this is a design choice. After issuing an invitation, the interface presents a button that copies a `base64` encoding of the invitation (including the private key) to the clipboard for secure out-of-band transmission. These encoded invitations are typically 45 characters long.[13] The recipient provides this input as an argument to the `/join` command in order to join a channel.

[13] ^ Additional characters may be added in sessions with dozens of outstanding invitations due to the variable-length encoding of the invitation identifier.

11.2.4 Natural Extensions

The secure messaging tool described in Section 11.2 is a proof-of-concept demonstration that can be expanded in several natural ways. This section describes some of these natural extensions.

A major limitation of the example application is that it does not support multiple devices sharing one account. Atwater et al. [AH16] discussed several standard mechanisms for adding multi-device support to secure messaging tools. There are generally two approaches that can be used in the example application: share the same Safehouse group state between devices, or have each device act as an independent Safehouse member. Sharing the group state between clients would require a mechanism to provision new devices with the existing group states (including the private states) for the channels in which the user is a member. This could be done by having clients periodically upload encrypted group states to the server for storage. The states could be encrypted using the locking key, *lsk* (see Section 11.2.1), which would allow new authorized devices to recover a recent state and catch up to the current group state by downloading a replay of events from the server. In the alternative approach where each device acts as an independent Safehouse member, the tool would need to be altered to support automated issuance of individual invitations. An old device would authorize a new device, issue individual invitations to its own identity public key in each channel, and then transmit the invitations—encrypted with *lsk*—to the new device.

The server software in the example application runs entirely as a single process on a single machine, which is not scalable. A simple extension would be to shard channels across multiple server machines. The servers could all share a single SQL-based backend, or the system could be made even more scalable by sharding payload and commit data across multiple databases. Due to the design of the system, it would also be easy to migrate channels across servers, because all state is stored in the database. This migration can be important if the channels handled by a machine become active enough to overload its resources, or if a machine is decommissioned.

Finally, the example application makes a variety of design decisions that might not be desirable even in the context of a similar IRC-like deployment. For example, the channel topic is stored in the confidential label-value store, but this is not necessary. The topic could be stored in the public label-value store instead, enabling the implementation of a channel listing command. The decision of whether or not the channel topic should be confidential could even be stored as a setting in the public label-value store, which is similar to channel modes in some IRC server software that hide the channel topic from the channel list. In general, Safehouse offers functionality that can replicate far more of the IRC experience than what the example application achieves (e.g., channel moderation, presence information, rich text formatting, and more).

11.3 Beyond IRC: Safehouse Integration Strategies

The IRC-like example application described in Section 11.2 demonstrates how to build a complete group secure messaging tool using Safehouse, but it is only one potential design. The high-level library described in Section 11.1 and the Safehouse protocol itself are far more widely applicable; in particular, Safehouse is not limited to desktop applications, global transcripts, IRC-like "server" and "channel" organization, or text-only communications. Safehouse can be used to implement tools that are suitable for all of the target scenarios listed in Chapter 7. This section briefly outlines some deployment techniques that could be used to replicate the functionality of popular group communication tools that currently lack end-to-end encryption.

A common communication scenario involves small groups assembled in an ad-hoc fashion, typically for event-based conversations. A tool for this scenario might take the form of a web application in which the user creates a new group conversation, a randomized link is generated and shared, the link allows others to immediately join the group, and the group is deleted as soon as the related event has concluded. A tool like this could initialize a Safehouse session for each group, configured with ephemeral authentication. The randomized link would contain the invitation data in the "fragment" component of the URL. In this scenario, each Safehouse session would be completely independent, with no data shared between them in the higher-level protocol.

Another target scenario involves creating a user experience similar to the popular Discord application. When using Discord, users invite others to join their "server" by sharing an invitation URL with them. Each "server" is created by a user that has administrative privileges as the "owner". Each "server" contains multiple "channels", including text-only channels and audio-based channels. All users in a server can view the contents of all text-based channels at all times (although channels can be locally "muted" in the interface to hide them), but audio-based channels must be joined explicitly. Consequently, the list of users is associated with the "server" and not with individual text-based channels. Discord also has a variety of other features, including customizable nicknames and colors, user taglines, granular moderation access policies, custom "emote" images, message reactions, and many monetization features. Moreover, Discord users can maintain contact lists and directly message other users outside of the context of a "server".

A secure group messaging tool that attempts to mimic the functionality of Discord using Safehouse would likely establish a single Safehouse session for all of the text-based channels on the "server". The customizable user attributes and "emote" images could be stored in the confidential

label-value store to hide them from the adversary.[14] The moderation privileges assigned to members could be stored in the public label-value store, with an appropriately configured group policy that ensures that members have the required privileges before performing operations. An array of identifiers for the text-based channels would likely be stored in the public label-value store in order to facilitate the expression of the moderation policy, but the human-readable names of the channels would be stored in the confidential label-value store. Payloads for text-based messages would include a channel identifier in the public tags, which would enable server-side support for "muting" channels to reduce bandwidth use by selectively routing payloads in a custom message router. Most of the rich text experience afforded by Discord, including reactions and file sharing, can be treated in the same way as standard message payloads from a cryptographic perspective. Direct messaging support outside of a "server" context could be achieved using an independent Safehouse session in a similar fashion to the aforementioned "ad-hoc group" scenario. Audio-based channels could be implemented securely using an independent Safehouse session with an unlimited-use bearer invitation that is stored in the confidential label-value store of the "server" Safehouse session. This would allow any member of the "server" to immediately join the audio-based "channel". Payloads in the audio-based channel would be unreliably transmitted audio fragments using the same techniques as Discord itself, but encrypting the audio data with a key derived from the group key for the "channel".

Slack is similar to Discord in many ways, but it places greater emphasis on private channels with many fewer members than the "server" itself. Consequently, many of the same techniques described above in the context of Discord could be used to implement a secure version of Slack, except that channels would be implemented as independent Safehouse sessions. Similarly to the Discord example, bearer invitations for these channels could be stored in the confidential label-value store for a Safehouse session associated with the "server", allowing members to immediately join certain channels.

Safehouse can also be used to implement tools for secure group video conferencing. One of the most difficult challenges for video conferencing tools is scalability. Individual clients normally have very limited bandwidth—particularly in the upstream direction for residential Internet connections—and potentially significant packet loss. Historically, video conferencing tools have relied on a central server to transcode the incoming video streams to lower bitrates in order to accommodate participants with low-bandwidth connections. Some tools go further and perform video compositing within the server, visually combining multiple video streams into a single stream before sending it to the participants. These efforts are frustrated by end-to-end encryption,

[14] ^ Discord allows users to purchase access to server-specific emote images that can then be used in other servers or in direct messages. Efficiently providing this feature would require granting the Safehouse server access to the images. Storing the images in the confidential label-value store would limit its use to the members in that Safehouse session.

which prevents the server from performing transcoding, compositing, or other video processing. Luckily, it is possible to develop highly scalable video conferencing tools that are completely compatible with end-to-end encryption by using a *layered coding* technique such as Scalable Video Coding (SVC) [SMW07]. The widespread adoption of the Zoom video conferencing tool has shown that this type of encryption-compatible encoding technology can be deployed on a massive scale. Zoom uses a proprietary video coding technique that is similar to SVC [Zoom19, 10:46]. Put simply, systems like SVC encode each frame of the source video as a sequence of packets where the initial packets provide a rough approximation of the image, and subsequent packets provide additional accuracy after decoding. The packets that provide additional accuracy can be dropped in order to obtain a lower-quality stream with a lower bitrate. All of the packets can be encrypted (e.g., using a key derived from the group key in a Safehouse session) before being uploaded to a central server using a UDP-based protocol (potentially with added information in the application layer to handle packet loss and reordering). The central server can learn which encrypted packets correspond to each SVC quality layer by arranging for the client to attach unencrypted public tags to the packets, or by using a preestablished packet ordering scheme. The server can then relay the stream to the other participants in the session. For recipients that require lower-bitrate streams (e.g., due to a low-bandwidth connection), the server can simply omit the packets that provide additional decoding accuracy. This approach avoids the need for the server to transcode video streams. Additionally, the tool can limit the maximum bandwidth required to participate in a conversation, without resorting to server-side compositing, by performing speaker detection (i.e., identifying when a participant is speaking) on the client side and sending this signal to the server. Using the received speaker detection signals, the server can make informed decisions about which streams to relay to participants at any given moment. Zoom uses this design [BBG+20], albeit with a server-selected AES-GCM key for encryption of the packets by default. Safehouse can be used to replace the key selection process in order to implement a reliable, scalable, and end-to-end encrypted secure video conferencing tool.

Developers can use the design patterns described in this section and the features provided by Safehouse to implement modern secure group messaging tools for all of the scenarios described in Chapter 7, among others, in straightforward ways. Developing a tool for a given scenario using Safehouse primarily involves identifying how to establish and configure Safehouse sessions, what information to store in the public and confidential label-value stores, what information to store independently of the Safehouse sessions, and how to structure payload data.

11.4 Chapter Summary

This chapter described a prototype implementation of the Safehouse protocol and provided examples of how it can be used in practice to develop secure group messaging tools, including the presentation of a functional IRC-like tool that was implemented as part of this work. Section 11.1 described a high-level library that abstracts the Safehouse protocol in order to support rapid development of tools for common target scenarios. The high-level library makes some simplifying assumptions that makes it less general than Safehouse itself, but also easier to use. The prototype implementation also includes a low-level library that implements the full Safehouse protocol presented in Chapter 10. The libraries are not intended for production use, but they are sufficiently developed to serve as a reference for the design and to evaluate the practicality of Safehouse. Section 11.2 described the complete IRC-like example application, including a graphical desktop client that was depicted in Section 11.2.3. Finally, Section 11.3 described some techniques that could be used to build tools for the other target scenarios listed in Chapter 7. Overall, this chapter demonstrated that the Safehouse protocol can be used to implement a practical application with a familiar user experience that provides very strong security properties without exposing the user to the substantial complexity of the underlying cryptography.

CHAPTER 12 | Conclusion

This dissertation introduced several new protocols that solve difficult cryptographic challenges at the core of secure messaging schemes. In particular, the new protocols establish shared secret keys to use for end-to-end encryption in a manner that provides deniability and protection against attacks by malicious insiders. These protocols can be used by developers to implement the majority of the conversation security design layer in a secure messaging tool while achieving strong security properties and non-interactive communication. The developers remain responsible for the application-specific aspects of the conversation security layer, such as how to use the established shared secret keys to encrypt messages, and the design of the surrounding protocol that triggers the exchange of keys and messages.

The thesis statement given in Section 1.1 claimed that it is possible to design efficient key exchange protocols that support asynchronous communication with insider security in typical two-party and group communication settings while relying on only common security assumptions. The new protocols that were introduced in this dissertation achieve these goals, and the performance evaluations performed on the prototype implementations show that the protocols are efficient enough for practical use. Specifically, the thesis statement is supported by the following observations:

- Chapter 5 introduced new DAKEs including DAKEZ (in Section 5.4) and XZDH (in Section 5.6). Both of these protocols provide online deniability, which is a form of insider security, in two-party secure messaging scenarios. XZDH can be used to implement secure messaging tools with non-interactive message delivery, which is suitable for common mobile applications. In particular, XZDH can be used as a drop-in replacement for X3DH in Signal [Ope13], one of the most popular secure messaging tools.

- Both DAKEZ and XZDH rely only on common security assumptions (see Proposition 1 and Theorem 3 for DAKEZ and Proposition 4 and Theorem 6 for XZDH): the instantiations described in Section 5.8 rely on DDH in a standard elliptic curve group, the security of AES, and the security of SHA-3. Composable security proofs for the DAKEs in the ROM were provided in Chapter 6.

- The performance evaluation of DAKEZ and XZDH presented in Section 5.9 shows that the new protocols achieve competitive performance compared to existing DAKEs while providing stronger deniability properties. The new protocols add approximately 1 ms of time and a few hundred bytes of communication to complete the key exchange.

- Part III introduced the Safehouse protocol. As described in Chapter 7, this new CGKA-like protocol manages shared keys throughout the lifetime of a secure group messaging session while providing strong security properties. The protocol makes extensive use of NIZKPKs to ensure that malicious insiders cannot deviate from the protocol. This insider security property becomes more important for larger groups where the trust between users is not as strong. Safehouse supports non-interactive secure group messaging protocols that can be used in most common communication scenarios, including the ones mentioned in the thesis statement.

- Safehouse must be instantiated with a particular BRAKEM construction, as discussed in Chapter 8. Section 8.2 introduced the $\text{BRAKEM}_{\star}^{\text{DDL}}$ construction. Theorem 7 claims that the security of $\text{BRAKEM}_{\star}^{\text{DDL}}$ relies on the DDH assumption and two unusual security assumptions in a standard "finite field DH" group. Section 8.4 provided a security proof of this claim. The two non-standard assumptions are the "discrete logarithm hidden subgroup" assumption (see Definition 1) and "multiplicative subgroup rounding with verifiers" assumption (see Definition 2). There are good reasons to believe that these assumptions hold (see Section 8.4.2.2.1), but they are not common assumptions. Section 8.5 introduced the $\text{BRAKEM}^{\text{ZK}}$ construction, which is an alternative to $\text{BRAKEM}_{\star}^{\text{DDL}}$. $\text{BRAKEM}^{\text{ZK}}$ relies on common zk-SNARK security assumptions that are used in popular cryptocurrencies. While cryptographers may have different opinions about the safety of the security assumptions for the two constructions, the security assumptions used by $\text{BRAKEM}^{\text{ZK}}$ satisfy the requirements of the thesis statement.

- Section 10.13 provided a performance evaluation of the Safehouse protocol. An experiment was performed that evaluated the additional time and communication size required to secure partially synthetic IRC chat logs using Safehouse. The results show that Safehouse can be considered efficient for groups of various sizes: a small group with approximately 50 members required hundreds of KiB of additional communication and milliseconds of computation per day, while a large group with approximately 1,400 members required tens of MiB of additional communication and seconds of computation per day. In addition to these synthetic benchmarks, Chapter 11 described a complete secure group messaging tool that was built with Safehouse in order to demonstrate its functionality. This tool can provide the experience of interacting with a Safehouse-based protocol in order to evaluate its efficiency from a subjective end-user perspective.

- The security properties provided by DAKEZ, XZDH, and Safehouse are designed to meet the needs of both "low-risk" and "high-risk" users as defined by Ermoshina et al. [EHM17; HEM18] and discussed in Section 2.1.2. The protocols all protect against global passive adversaries by securely establishing a shared secret key, and against server seizures by giving servers no secrets to keep. The protocols all provide anonymity preservation, allowing them to be combined with a private transport layer to protect against metadata leak. Moreover, the protocols can be used to develop tools that do not share metadata with insiders (e.g., by using Safehouse with only ephemeral authentication), provide local protections against device seizure (e.g., by automatically deleting messages and protecting local data with a password, as with the demonstration application in Chapter 11), and provide paths to recovery after device seizure (e.g., by using Safehouse's post-compromise security and invitation features), which are all significant concerns for high-risk users. In general, the protocols were designed in response to all of the findings from the usability studies discussed in Section 2.1.

Much work remains to be done to improve the deployability and efficiency of Safehouse. The chapters in Part III introduced Safehouse, described the prototype implementation, presented the results of a performance evaluation, and showcased a complete proof-of-concept secure group messaging tool built using the protocol. However, while Chapter 7 claimed that Safehouse provides a variety of security properties and Section 8.4 proved the security of BRAKEM$^{\text{DDL}}_{\star}$ (the most significant building block for Safehouse), no security proof for the overall protocol was given. This is the largest outstanding issue that should be resolved before Safehouse is deployed in real secure group messaging tools. Unfortunately, as showcased by the security proofs for the new DAKEs provided in Chapter 6, formally proving that complex protocols are secure is a very challenging task. Safehouse is far larger than any of the protocols introduced in Part II, and verifying its claimed security properties will require a correspondingly larger security proof. Safehouse has been designed to anticipate this future work by breaking it down into smaller components. In particular, composable security proofs (e.g., using the GUC framework) could be produced for each of the *operations* described in Chapter 10. For each operation, the proofs would need to show that the resulting key graph always accurately reflects the new group state, even in the presence of an active adversary, and that the security of the blinded identities and the confidential label-value store is maintained. Formalizing the required properties and providing the required security proofs is left as the most significant future work.

In addition to proving the security of Safehouse, some methods for improving the efficiency of the protocol are immediately apparent. The performance optimizations for BRAKEM$^{\text{DDL}}_{\star}$ described in Section 8.3.2 may be improved by investigating alternative integer representations or using new x86-64 instruction set extensions like AVX-512 IFMA [DG19]. Experimental results in the context of homomorphic encryption [BKS+21, Tbl. 4] and quantum-resistant KEMs [GK19] have suggested that implementing modular multiplication with AVX-512 IFMA may speed up

computation times by a factor of 1.7. In addition to the x86-64 optimizations, an entirely different set of optimizations should be used to implement BRAKEM$^{\text{DDL}}_{\star}$ for ARM-based processors, which are typically found in mobile devices. An implementation of BRAKEM$^{\text{ZK}}$ as described in Section 8.5 was developed and partially evaluated in Section 8.6, but fully implementing Safehouse using BRAKEM$^{\text{ZK}}$ and evaluating the resulting communication sizes (as with BRAKEM$^{\text{DDL}}_{\star}$ in Section 10.13) is left to future work. Given the rapid pace of development of new zero-knowledge proof systems, new techniques may be able to implement BRAKEM even more efficiently in the near future. Section 10.8.5 described how the Safehouse authentication mechanism could be implemented using Borromean ring signatures, but implementing and evaluating this technique is left to future work. Although Section 10.11 introduced several key control schemes and the performance evaluation in Section 10.13 provided some initial intuition about parameter selection, more work is required to discover optimal algorithms and configurations for a broader range of specific deployment scenarios.

Our reliance on Internet-based group communication technologies has grown rapidly, and it will likely continue to do so. At the same time, the complexity of the conversation security design layer for group communication has caused secure messaging protocols to lag behind the demand for new tools. It is the author's hope that the new protocols presented in this dissertation will inform the next generation of secure group messaging tools by helping to solve the core cryptographic challenges. The time to deploy efficient secure group messaging protocols with strong security properties is upon us.

Concurrent Work

While this dissertation was written, secure group messaging was the subject of many active research projects from a variety of sources, including academia, the private sector, and the open-source community. This heightened attention is a testament to the importance of secure group messaging technologies and to the absence of a secure, scalable, and suitably featureful solution. Work on the Safehouse protocol, described in Part III, began in late 2017, approximately near the time that the ART protocol [CCG+18] was being developed. As discussed in Chapter 4, ART was the project that heralded the emergence of CGKAs, eventually culminating in the MLS project organized under the auspices of the IETF.

Safehouse was designed in intentional isolation from developments in the MLS project. The rationale for this approach was that it would result in a truly independent construction free of any design choices or assumptions made by the MLS project; if the solutions diverged, then it would provide an opportunity to combine the best ideas from both efforts, and it would also highlight potential design variables in the solution space. The reader likely has access to hindsight that the author does not, with which they may judge the wisdom of this strategy. In any event, it is interesting that the MLS project subsequently and independently developed extensions that correspond to components of Safehouse, such as the near equivalence between TTKEM's "blanked nodes" and Safehouse's "superfluous edges". This convergence hints that the solution space for the requirements may be small, and that the chosen approach may be a good one.

The author is aware of several independent and concurrent research efforts that published relevant technical reports while this dissertation was being finalized. The remainder of this section provides only a cursory summary of these reports; interested readers are encouraged to consult the resulting publications to compare the results in greater detail.

Goel et al. [GGHK21] recognized that Borromean ring signatures [MP15] could be generalized in the manner described in Section 10.8.5. They provided rigorous definitions and security proofs for the generalized construction. Poettering et al. [PRSS21] surveyed and systematized GKEs and CGKAs with formal computational game-based security models, which includes many of the schemes surveyed in Chapters 3 and 4. Their systematization includes a newly proposed model that is intended to unify the desirable properties of a modern GKE without being specific to any particular protocol. Analyzing TKLL and Safehouse using this new model is a promising direction for future work. Finally, Alwen et al. [AJM20] introduced Insider Secure TreeKEM (ITK), a variant of TreeKEM (see Sections 4.2 and 4.5) that aims to provide insider security. Unlike Safehouse, which uses NIZKPKs to prove that all modifications to the group state were performed correctly and that the appropriate private keys were transferred to the expected

recipients, the approach taken by ITK is much simpler: in essence, ITK is a variant of the MLS protocol in which each member produces a digital signature attesting that its path in the key graph accurately reflects the members' knowledge of the private keys. This simpler approach provides much weaker guarantees: the group state becomes secure only when all malicious insiders are evicted. Safehouse provides stronger guarantees: payloads become confidential only when all malicious insiders are evicted, but the entire group state is always guaranteed to be consistent and correctly constructed, even when malicious insiders remain in the session.

Letters of Copyright Permission

Artwork Licenses

This work contains four commissioned artworks, all licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA. A copy of this license is also included in the next section.

The contents of the listed pages below are copyrighted by the named licensors:

- Page xxiv: Copyright © 2021 Brandon Palas (brandonpalas.com). Some rights reserved. Used under CC BY 4.0. Modifications: color changes, placement adjustments, and added text.
- Page 10: Copyright © 2021 Lukáš Vašut (lukasvasut.myportfolio.com). Some rights reserved. Used under CC BY 4.0. Modifications: blurred particles, cropped, and downscaled.
- Page 71: Copyright © 2021 Nicole Cardiff (www.artofnicolecardiff.com). Some rights reserved. Used under CC BY 4.0. Modifications: downscaled.
- Page 162: Copyright © 2021 Tower Junkie Art. Some rights reserved. Used under CC BY 4.0. Modifications: lighting changes, cropped, and downscaled.

Creative Commons Attribution 4.0 International License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1—Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.
c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.
d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.
e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.
f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.
g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.
i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.
j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.
k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2—Scope.

a. License grant.
   1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
      a. reproduce and Share the Licensed Material, in whole or in part; and
      b. produce, reproduce, and Share Adapted Material.
   2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.
   3. Term. The term of this Public License is specified in Section 6(a).
   4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.
   5. Downstream recipients.
      a. Offer from the Licensor—Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
      b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.
   6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).
b. Other rights.
   1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
   2. Patent and trademark rights are not licensed under this Public License.
   3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3—License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.
   1. If You Share the Licensed Material (including in modified form), You must:
      a. retain the following if it is supplied by the Licensor with the Licensed Material:
         i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
         ii. a copyright notice;
         iii. a notice that refers to this Public License;
         iv. a notice that refers to the disclaimer of warranties;
         v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
      b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
      c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
   2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
   3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
   4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4—Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and
c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5—Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.
b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.
c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6—Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:
   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
   2. upon express reinstatement by the Licensor.
   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.
c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7—Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8—Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.
c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Font Licenses

This document was typeset using LaTeX 2$_\varepsilon$. The body text is set in the Bitstream Charter ® typeface. Headings are set in the Biolinum typeface. Monospaced text is set in the `Bera Mono` typeface, which is derived from the Bitstream Vera ® fonts. Latin text is set in the CINZEL typeface. Section category icons are from the Font Awesome Free framework. This document contains embedded subsets of these fonts. The licenses for these fonts are included in the following sections.

Bitstream Charter License

Biolinum and Cinzel License

The OFL allows the licensed fonts to be used, studied, modified and redistributed freely as long as they are not sold by themselves. The fonts, including any derivative works, can be bundled, embedded, redistributed and/or sold with any software provided that any reserved names are not used by derivative works. The fonts and derivatives, however, cannot be released under any other type of license. The requirement for fonts to remain under this license does not apply to any document created using the fonts or their derivatives.

DEFINITIONS
"Font Software" refers to the set of files released by the Copyright Holder(s) under this license and clearly marked as such. This may include source files, build scripts and documentation. "Reserved Font Name" refers to any names specified as such after the copyright statement(s). "Original Version" refers to the collection of Font Software components as distributed by the Copyright Holder(s). "Modified Version" refers to any derivative made by adding to, deleting, or substituting—in part or in whole—any of the components of the Original Version, by changing formats or by porting the Font Software to a new environment. "Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS
Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions:
1) Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself.
2) Original or Modified Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.
3) No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.
4) The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.
5) The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION
This license becomes null and void if any of the above conditions are not met.

DISCLAIMER
THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Bera License

Copyright © 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

Font Awesome Free License

This document contains unmodified images from Font Awesome Free 5.15.3 by @fontawesome—https://fontawesome.com/. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

References

[AAA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NIST CSRC, 2020. DOI: 10.6028/NIST.IR.8309. *(Three citations on pages 93, 103, and 128.)*

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 5–17. DOI: 10.1145/2810103.2813707. *(Two citations on pages 206 and 207.)*

[ABH+15] Erinn Atwater, Cecylia Bocovich, Urs Hengartner, Ed Lank, and Ian Goldberg. "Leading Johnny to Water: Designing for Usability and Trust". In: *Symposium On Usable Privacy and Security (SOUPS)*. USENIX Association, 2015, pp. 69–88. *(Two citations on pages 13 and 15.)*

[ABM+03] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott Vanstone. "Validation of Elliptic Curve Public Keys". In: *Public Key Cryptography (PKC)*. Springer, 2003, pp. 211–223. DOI: 10.1007/3-540-36288-6_16. *(One citation on page 102.)*

[ACC+19] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. *Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement*. Tech. rep. 2019/1489. Cryptology ePrint Archive, 2019. URL: https://eprint.iacr.org/2019/1489 (visited on 2021-04-24). *(Five citations on pages 55, 65, 67, and 68.)*

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *Advances in Cryptology—CRYPTO*. Springer. 2020, pp. 248–277. DOI: 10.1007/978-3-030-56784-2_9. *(One citation on page 68.)*

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. "Continuous Group Key Agreement with Active Security". In: *Theory of Cryptography*. Springer, 2020, pp. 261–290. DOI: 10.1007/978-3-030-64378-2_10. *(Six citations on pages 3, 65, 66, 68, and 168.)*

[ACMP10] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. "Flexible Group Key Exchange with On-Demand Computation of Subgroup Keys". In: *Progress in Cryptology—AFRICACRYPT* 10 (2010), pp. 351–368. DOI: 10.1007/978-3-642-12678-9_21. *(One citation on page 47.)*

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange—A New Hope". In: *Security Symposium*. USENIX, 2016, pp. 327–343. *(Two citations on pages 104 and 128.)*

[AG07] Chris Alexander and Ian Goldberg. "Improved User Authentication in Off-The-Record Messaging". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2007, pp. 41–47. DOI: 10.1145/1314333.1314340. *(Two citations on pages 23 and 107.)*

[AGJS13] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. *Innovative Technology for CPU Based Attestation and Sealing*. White paper. Intel, 2013. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/hasp-2013-innovative-technology-for-attestation-and-sealing-413939.pdf (visited on 2021-04-24). *(One citation on page 77.)*

[AH16] Erinn Atwater and Urs Hengartner. "Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices". In: *Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2016, pp. 91–102. DOI: 10.1145/2939918.2939932. *(Three citations on pages 29, 102, and 497.)*

[AHR05] Ben Adida, Susan Hohenberger, and Ronald L. Rivest. "Ad-Hoc-Group Signatures from Hijacked Keypairs". In: *DIMACS Workshop on Theft in E-Commerce*. 2005. *(One citation on page 89.)*

[AJM20] Joël Alwen, Daniel Jost, and Marta Mularczyk. *On The Insider Security of MLS*. Tech. rep. 2020/1327. Cryptology ePrint Archive, 2020. URL: https://eprint.iacr.org/2020/1327 (visited on 2021-04-24). *(One citation on page 506.)*

[AKP+17] Ruba Abu-Salma, Kat Krol, Simon Parkin, Victoria Koh, Kevin Kwan, Jazib Mahboob, Zahra Traboulsi, and M. Angela Sasse. "The Security Blanket of the Chat World: An Analytic Evaluation and a User Study of Telegram". In: *European Workshop on Usable Security (EuroUSEC)*. 2017. DOI: 10.14722/eurousec.2017.23006. *(One citation on page 15.)*

[AM93] A. O. L. Atkin and François Morain. "Elliptic curves and primality proving". In: *Mathematics of Computation* 61.203 (1993), pp. 29–68. DOI: 10.1090/S0025-5718-1993-1199989-X. *(One citation on page 209.)*

[AMPS18] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. "Prime and Prejudice: Primality Testing Under Adversarial Conditions". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2018, pp. 281–298. DOI: 10.1145/3243734.3243787. *(One citation on page 209.)*

[An17] Daniel An. *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed*. 2017. URL: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/ (visited on 2021-04-24). *(Two citations on pages 467 and 469.)*

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. "1-out-of-*n* Signatures from a Variety of Keys". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2002, pp. 415–432. DOI: 10.1007/3-540-36178-2_26. *(One citation on page 89.)*

[ARA+19] Brooke Auxier, Lee Rainie, Monica Anderson, Andrew Perrin, Madhu Kumar, and Erica Turner. *Americans and Privacy: Concerned, Confused and Feeling Lack of Control Over Their Personal Information*. 2019. URL: https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/ (visited on 2021-04-24). *(One citation on page 1.)*

[Arc18] Scott Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*. Internet Draft. IETF, 2018. URL: https://tools.ietf.org/html/draft-irtf-cfrg-xchacha (visited on 2021-04-24). *(Two citations on pages 478 and 487.)*

[ARUW18] Ruba Abu-Salma, Elissa M Redmiles, Blase Ur, and Miranda Wei. "Exploring User Mental Models of End-to-End Encrypted Communication Tools". In: *Workshop on Free and Open Communications on the Internet (FOCI)*. USENIX. 2018. *(One citation on page 15.)*

[ASB+17] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. "Obstacles to the Adoption of Secure Communication Tools". In: *Symposium on Security and Privacy (S&P)*. IEEE. 2017. DOI: 10.1109/SP.2017.65. *(Two citations on page 15.)*

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *Symposium on Security and Privacy (S&P)*. IEEE. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020. *(One citation on page 271.)*

[BBB94] François Bergeron, Jean Berstel, and Srecko Brlek. "Efficient computation of addition chains". In: *Journal de Théorie des Nombres de Bordeaux* 6.1 (1994), pp. 21–38. DOI: 10.5802/jtnb.104. *(One citation on page 232.)*

[BBBD89] François Bergeron, Jean Berstel, Srecko Brlek, and Christine Duboc. "Addition Chains Using Continued Fractions". In: *Journal of Algorithms* 10.3 (1989), pp. 403–412. DOI: 10.1016/0196-6774(89)90036-9. *(One citation on page 232.)*

[BBC+13] Naomi Benger, David Bernhard, Dario Catalano, Manuel Charlemagne, David Conti, Biljana Cubaleska, Hernando Fernando, Dario Fiore, Steven Galbraith, David Galindo, Jens Hermans, Vincenzo Iovino, Tibor Jager, Markulf Kohlweiss, Benoit Libert, Richard Lindner, Hans Loehr, Danny Lynch, Richard Moloney, Khaled Ouafi, Benny Pinkas, Frantisek Polach, Mario Di Raimondo, Markus Rückert, Michael Schneider, Vijay Singh, Nigel Smart, Martijn Stam, Fré Vercauteren, Jorge Villar Santos, and Steve Williams. *Final Report on Main Computational Assumptions in Cryptography*. Ed. by Fré Vercauteren. Tech. rep. 2013. URL: https://www.ecrypt.eu.org/ecrypt2/documents/D.MAYA.6.pdf (visited on 2021-04-24). *(One citation on page 272.)*

[BBF+19] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange". In: *Post-Quantum Cryptography*. Springer, 2019, pp. 206–226. DOI: 10.1007/978-3-030-25510-7_12.
(Two citations on pages 82 and 128.)

[BBG+20] Josh Blum, Simon Booth, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. "E2E Encryption for Zoom Meetings. Version 3". 2020. URL: https://github.com/zoom/zoom-e2e-whitepaper (visited on 2021-04-24).
(Two citations on pages 1 and 500.)

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Scalable Zero Knowledge with No Trusted Setup". In: *Advances in Cryptology—CRYPTO*. Springer. 2019, pp. 701–732. DOI: 10.1007/978-3-030-26954-8_23.
(One citation on page 271.)

[BBJ+08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. "Twisted Edwards Curves". In: *Progress in Cryptology—AFRICACRYPT*. Springer. 2008, pp. 389–405. DOI: 10.1007/978-3-540-68164-9_26.
(Two citations on pages 275 and 281.)

[BBM+20] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol Draft 09*. Internet Draft. IETF, 2020. URL: https://tools.ietf.org/html/draft-ietf-mls-protocol-09 (visited on 2021-04-24).
(Seven citations on pages 2, 3, 5, 40, 61, 62, and 325.)

[BBM09] Timo Brecher, Emmanuel Bresson, and Mark Manulis. "Fully Robust Tree-Diffie-Hellman Group Key Exchange". In: *International Conference on Cryptology and Network Security (CANS)*. Springer. 2009, pp. 478–497. DOI: 10.1007/978-3-642-10433-6_33.
(Four citations on pages 51, 185, and 203.)

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups*. Tech. rep. Inria Paris, 2018. URL: https://hal.univ-rennes2.fr/INRIA/hal-02425247 (visited on 2021-04-24).
(Three citations on pages 57, 61, and 67.)

[BC04] Emmanuel Bresson and Dario Catalano. "Constant Round Authenticated Group Key Agreement via Distributed Computation". In: *Public Key Cryptography (PKC)*. Springer. 2004, pp. 115–129. DOI: 10.1007/978-3-540-24632-9_9.
(One citation on page 44.)

[BCC+15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. "How to Manipulate Curve Standards: A White Paper for the Black Hat". In: *Security Standardisation Research*. Springer. 2015, pp. 109–139. DOI: 10.1007/978-3-319-27152-1_6.
(Two citations on pages 210 and 211.)

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again". In: *Innovations in Theoretical Computer Science*. ACM. 2012, pp. 326–349. DOI: 10.1145/2090236.2090263. *(One citation on page 273.)*

[BCP01] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. "Provably Authenticated Group Diffie-Hellman Key Exchange—The Dynamic Case". In: *Advances in Cryptology—ASIACRYPT*. Vol. 2248. Springer. 2001, pp. 290–309. DOI: 10.1007/3-540-45682-1_18. *(One citation on page 45.)*

[BCP02] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. "Dynamic Group Diffie-Hellman Key Exchange under Standard Assumptions". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2002, pp. 321–336. DOI: 10.1007/3-540-46035-7_21. *(One citation on page 45.)*

[BD94] Mike Burmester and Yvo Desmedt. "A Secure and Efficient Conference Key Distribution System". In: *Workshop on the Theory and Application of Cryptographic Techniques*. Springer. 1994, pp. 275–286. DOI: 10.1007/BFb0053443. *(Four citations on pages 46, 294, 305, and 322.)*

[BD96] Mike Burmester and Yvo Desmedt. "Efficient and Secure Conference-Key Distribution". In: *International Workshop on Security Protocols*. Springer. 1996, pp. 119–129. DOI: 10.1007/3-540-62494-5_12. *(Two citations on pages 48 and 49.)*

[BDG15] Nikita Borisov, George Danezis, and Ian Goldberg. "DP5: A Private Presence Service". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2015.2 (2015), pp. 4–24. DOI: 10.1515/popets-2015-0008. *(One citation on page 29.)*

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31. *(One citation on page 487.)*

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89. DOI: 10.1007/s13389-012-0027-1. *(Four citations on pages 104, 105, and 487.)*

[BDPR98] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. "Relations Among Notions of Security for Public-Key Encryption Schemes". In: *Advances in Cryptology—CRYPTO*. Springer. 1998, pp. 26–45. DOI: 10.1007/BFb0055718. *(One citation on page 146.)*

[BDS03] Rana Barua, Ratna Dutta, and Palash Sarkar. "Extending Joux's Protocol to Multi Party Key Agreement". In: *Progress in Cryptology—INDOCRYPT*. Vol. 2904. Springer. 2003, pp. 205–217. DOI: 10.1007/978-3-540-24582-7_15. *(One citation on page 51.)*

[Ber02] Daniel J. Bernstein. "Pippenger's Exponentiation Algorithm". 2002. URL: https://cr.yp.to/papers/pippenger.pdf (visited on 2021-04-24).
(Three citations on pages 234 and 238.)

[Ber04] Daniel J. Bernstein. "Distinguishing prime numbers from composite numbers: the state of the art in 2004". 2004. URL: https://cr.yp.to/primetests.html (visited on 2021-04-24). *(One citation on page 209.)*

[Ber06] Daniel J. Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *Public Key Cryptography (PKC)*. Springer, 2006, pp. 207–228. DOI: 10.1007/11745853_14.
(Eight citations on pages 105, 219, 297, 307, 312, and 370.)

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. "Off-the-Record Communication, or, Why Not To Use PGP". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2004, pp. 77–84. DOI: 10.1145/1029179.1029200. *(Eight citations on pages 2, 19, 21–23, 31, and 32.)*

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2003, pp. 416–432. DOI: 10.1007/3-540-39200-9_26. *(One citation on page 89.)*

[BGMW93] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. "Fast Exponentiation with Precomputation. (Extended Abstract)". In: *Advances in Cryptology—EUROCRYPT*. Springer, 1993, pp. 200–207. DOI: 10.1007/3-540-47555-9_18.
(Two citations on page 235.)

[BGMW95] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. "Fast Exponentiation with Precomputation: Algorithms and Lower Bounds". 1995. URL: http://dbwilson.com/bgmw/ (visited on 2021-04-24). *(Three citations on pages 235 and 236.)*

[BGP20] Gautam Botrel, Gus Gutoski, and Thomas Piellard. *Introducing gnark: a fast zero-knowledge proof library*. 2020. URL: https://hackmd.io/@zkteam/gnark (visited on 2021-04-24).
(One citation on page 274.)

[BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. "Fast Batch Verification for Modular Exponentiation and Digital Signatures". In: *Advances in Cryptology—EUROCRYPT*. Springer, 1998, pp. 236–250. DOI: 10.1007/BFb0054130. *(One citation on page 396.)*

[BH77] Carter Bays and Richard H. Hudson. "The Segmented Sieve of Eratosthenes and Primes in Arithmetic Progressions up to 10^{12}". In: *BIT Numerical Mathematics* 17.2 (1977), pp. 121–127. DOI: 10.1007/BF01932283. *(One citation on page 213.)*

[BJN00] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. "Why Textbook ElGamal and RSA Encryption Are Insecure". In: *Advances in Cryptology—ASIACRYPT*. Springer, 2000, pp. 30–43. DOI: 10.1007/3-540-44448-3_3. *(One citation on page 258.)*

[BKM06] Adam Bender, Jonathan Katz, and Ruggero Morselli. "Ring Signatures: Stronger Definitions, and Constructions without Random Oracles". In: *Theory of Cryptography*. Springer, 2006, pp. 60–79. DOI: 10.1007/11681878_4. *(Seven citations on pages 89, 91, and 92.)*

[BKR00] Mihir Bellare, Joe Kilian, and Phillip Rogaway. "The Security of the Cipher Block Chaining Message Authentication Code". In: *Journal of Computer and System Sciences* 61.3 (2000), pp. 362–399. DOI: 10.1006/jcss.1999.1694. *(Four citations on pages 99, 101, 148, and 154.)*

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. *Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52*. Tech. rep. 2103.16400. arXiv, 2021. URL: https://arxiv.org/abs/2103.16400 (visited on 2021-04-24). *(One citation on page 504.)*

[BL14] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. 2014. URL: https://safecurves.cr.yp.to/ (visited on 2021-04-24). *(One citation on page 209.)*

[BLS03] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. "Constructing Elliptic Curves with Prescribed Embedding Degrees". In: *Security in Communication Networks*. Springer. 2003, pp. 257–267. DOI: 10.1007/3-540-36413-7_19. *(Two citations on pages 274 and 275.)*

[BMHD08] Zhang Bin, Feng Meng, Xiong Hou-ren, and Hu Dian-you. "Design and Implementation of Secure Instant Messaging System Based on MSN". In: *International Symposium on Computer Science and Computational Technology*. Vol. 1. IEEE. 2008, pp. 38–41. DOI: 10.1109/ISCSCT.2008.49. *(One citation on page 30.)*

[BMP04] Colin Boyd, Wenbo Mao, and Kenneth G. Paterson. "Key Agreement using Statically Keyed Authenticators". In: *Applied Cryptography and Network Security*. Springer, 2004, pp. 248–262. DOI: 10.1007/978-3-540-24852-1_18. *(One citation on page 23.)*

[BN03] Colin Boyd and Juan Manuel González Nieto. "Round-optimal Contributory Conference Key Agreement". In: *Public Key Cryptography (PKC)*. Vol. 3. Springer. 2003, pp. 161–174. DOI: 10.1007/3-540-36288-6_12. *(One citation on page 44.)*

[BN06] Mihir Bellare and Gregory Neven. "Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2006, pp. 390–399. DOI: 10.1145/1180405.1180453. *(Two citations on pages 307 and 308.)*

[Bou00] Fabrice Boudot. "Efficient Proofs that a Committed Number Lies in an Interval". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2000, pp. 431–444. DOI: 10.1007/3-540-45539-6_31. *(One citation on page 187.)*

[BP04] Mihir Bellare and Adriana Palacio. "The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols". In: *Advances in Cryptology—CRYPTO*. Springer. 2004, pp. 273–289. DOI: 10.1007/978-3-540-28628-8_17. *(Two citations on pages 272 and 273.)*

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. "Authenticated Key Exchange Secure Against Dictionary Attacks". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2000, pp. 139–155. DOI: 10.1007/3-540-45539-6_11. *(Six citations on pages 23, 81, 100, 107, and 127.)*

[BR93a] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution". In: *Advances in Cryptology—CRYPTO*. Springer, 1993, pp. 232–249. DOI: 10.1007/3-540-48329-2_21. *(Three citations on pages 23, 80, and 81.)*

[BR93b] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *Conference on Computer and Communications Security (CCS)*. ACM. 1993, pp. 62–73. DOI: 10.1145/168588.168596. *(One citation on page 116.)*

[Bra39] Alfred Brauer. "On Addition Chains". In: *Bulletin of the American Mathematical Society* 45 (1939), pp. 736–739. DOI: 10.1090/S0002-9904-1939-07068-7. *(Two citations on pages 228 and 237.)*

[Bri18] Darrell Bricker. *Majority (52%) says they're More Concerned about Online Privacy than they were a Year Ago*. 2018. URL: https://www.ipsos.com/en-ca/news-polls/Centre-for-International-Governance-Innovation-Online-Privacy-Poll-May-17-2018 (visited on 2021-04-24). *(One citation on page 1.)*

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. "Determining the Core Primitive for Optimally Secure Ratcheting". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2020, pp. 621–650. DOI: 10.1007/978-3-030-64840-4_21. *(One citation on page 32.)*

[BS03] Dan Boneh and Alice Silverberg. "Applications of Multilinear Forms to Cryptography". In: *Contemporary Mathematics* 324.1 (2003), pp. 71–90. DOI: 10.1090/conm/324/05731. *(One citation on page 42.)*

[BSS02] Emmanuel Bresson, Jacques Stern, and Michael Szydlo. "Threshold Ring Signatures and Applications to Ad-hoc Groups". In: *Advances in Cryptology—CRYPTO*. Springer. 2002, pp. 465–480. DOI: 10.1007/3-540-45708-9_30. *(One citation on page 89.)*

[BST07] Jiang Bian, Remzi Seker, and Umit Topaloglu. "Off-the-Record Instant Messaging for Group Conversation". In: *International Conference on Information Reuse and Integration (IRI)*. IEEE. 2007, pp. 79–84. DOI: 10.1109/IRI.2007.4296601. *(Two citations on pages 33 and 36.)*

[BW98] Klaus Becker and Uta Wille. "Communication Complexity of Group Key Distribution". In: *Conference on Computer and Communications Security (CCS)*. ACM. 1998, pp. 1–6. DOI: 10.1145/288090.288094. *(One citation on page 43.)*

[BZ14] Dan Boneh and Mark Zhandry. "Multiparty Key Exchange, Efficient Traitor Tracing, and More from Indistinguishability Obfuscation". In: *Advances in Cryptology—CRYPTO*. Springer. 2014, pp. 480–499. DOI: 10.1007/978-3-662-44371-2_27. *(One citation on page 42.)*

[Cam98] Jan Camenisch. "Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem". PhD thesis. ETH Zurich, 1998. DOI: 10.3929/ethz-a-001923735. *(Three citations on pages 188, 245, and 246.)*

[Can01] Ran Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *Symposium on Foundations of Computer Science*. IEEE. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888. *(Four citations on pages 113, 115, and 119.)*

[CBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. “Riposte: An Anonymous Messaging System Handling Millions of Users”. In: *Symposium on Security and Privacy (S&P)*. IEEE. 2015, pp. 321–338. DOI: 10.1109/SP.2015.27. *(One citation on page 37.)*

[CCD+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. “A Formal Security Analysis of the Signal Messaging Protocol”. In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27. *(One citation on page 32.)*

[CCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. “On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees”. In: *Conference on Computer and Communications Security (CCS)*. ACM. 2018, pp. 1802–1819. DOI: 10.1145/3243734.3243747. *(Five citations on pages 2, 55, 203, and 506.)*

[CDF+07] John Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. *OpenPGP Message Format*. RFC 4880. IETF, 2007, pp. 1–90. URL: https://tools.ietf.org/html/rfc4880 (visited on 2021-04-24). *(One citation on page 30.)*

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. “Universally Composable Security with Global Setup”. In: *Theory of Cryptography*. Springer. 2007, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4. *(Four citations on pages 113 and 114.)*

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. “Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols”. In: *Advances in Cryptology—CRYPTO*. Springer. 1994, pp. 174–187. DOI: 10.1007/3-540-48658-5_19. *(Three citations on pages 90, 394, and 403.)*

[CEG88] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. “An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations”. In: *Advances in Cryptology—EUROCRYPT*. Springer. 1988, pp. 127–141. DOI: 10.1007/3-540-39118-5_13. *(One citation on page 185.)*

[CF10] Henry Corrigan-Gibbs and Bryan Ford. “Dissent: Accountable Anonymous Group Messaging”. In: *Conference on Computer and Communications Security (CCS)*. ACM. 2010, pp. 340–350. DOI: 10.1145/1866307.1866346. *(One citation on page 37.)*

[CFA+05] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Scientific ed. by Henri Cohen and Gerhard Frey. Executive ed. by Christophe Doche. DOI (EBOOK): 10.1201/9781420034981. Chapman & Hall/CRC, 2005. ISBN: 1584885181. *(Two citations on page 232.)*

[CFT98] Agnes Chan, Yair Frankel, and Yiannis Tsiounis. “Easy Come — Easy Go Divisible Cash”. In: *Advances in Cryptology—EUROCRYPT*. Springer. 1998, pp. 561–575. DOI: 10.1007/BFb0054154. *(Two citations on pages 187 and 188.)*

[CFZ14] Sherman S. M. Chow, Matthew Franklin, and Haibin Zhang. “Practical Dual-Receiver Encryption”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2014, pp. 85–105. DOI: 10.1007/978-3-319-04852-9_5. *(Eight citations on pages 85, 86, and 88.)*

[CGM+11] Sandy Clark, Travis Goodspeed, Perry Metzger, Zachary Wasserman, Kevin Xu, and Matt Blaze. “Why (Special Agent) Johnny (Still) Can’t Encrypt: A Security Analysis of the APCO Project 25 Two-way Radio System”. In: *Security Symposium*. USENIX, 2011. *(One citation on page 13.)*

[Cha81] David Chaum. “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms”. In: *Communications of the ACM* 24.2 (1981), pp. 84–90. DOI: 10.1145/358549.358563. *(One citation on page 37.)*

[Cha90] David Chaum. “Zero-Knowledge Undeniable Signatures (extended abstract)”. In: *Advances in Cryptology—EUROCRYPT*. Springer. 1990, pp. 458–464. DOI: 10.1007/3-540-46877-3_41. *(One citation on page 203.)*

[CHK03] Ran Canetti, Shai Halevi, and Jonathan Katz. “A Forward-Secure Public-Key Encryption Scheme”. In: *Advances in Cryptology—EUROCRYPT*. Springer, 2003, pp. 255–271. DOI: 10.1007/3-540-39200-9_16. *(One citation on page 31.)*

[CK02a] Ran Canetti and Hugo Krawczyk. “Security Analysis of IKE’s Signature-based Key-Exchange Protocol”. In: *Advances in Cryptology—CRYPTO*. Springer. 2002, pp. 143–161. DOI: 10.1007/3-540-45708-9_10. *(Three citations on pages 82, 96, and 120.)*

[CK02b] Ran Canetti and Hugo Krawczyk. “Universally Composable Notions of Key Exchange and Secure Channels”. In: *Advances in Cryptology—EUROCRYPT*. Springer. 2002, pp. 337–351. DOI: 10.1007/3-540-46035-7_22. *(Ten citations on pages 80, 81, 115, and 118–120.)*

[CKFP10] Joseph A. Cooley, Roger I. Khazan, Benjamin W. Fuller, and Galen E. Pickard. “GROK: A Practical System for Securing Group Communications”. In: *International Symposium on Network Computing and Applications*. IEEE. 2010, pp. 100–107. DOI: 10.1109/NCA.2010.20. *(One citation on page 33.)*

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. “Secure and Efficient Asynchronous Broadcast Protocols”. In: *Advances in Cryptology—CRYPTO*. Springer. 2001, pp. 524–541. DOI: 10.1007/3-540-44647-8_31. *(One citation on page 294.)*

[CM10] Qingfeng Cheng and Chuangui Ma. *Security Weakness of Flexible Group Key Exchange with On-Demand Computation of Subgroup Keys*. Tech. rep. 1008.1221. arXiv, 2010. URL: https://arxiv.org/abs/1008.1221 (visited on 2021-04-24). *(One citation on page 48.)*

[CM99a] Jan Camenisch and Markus Michels. "Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes". In: *Advances in Cryptology—EUROCRYPT*. Springer, 1999, pp. 107–122. DOI: 10.1007/3-540-48910-X_8. *(One citation on page 183.)*

[CM99b] Jan Camenisch and Markus Michels. "Separability and Efficiency for Generic Group Signature Schemes". In: *Advances in Cryptology—CRYPTO*. Springer. 1999, pp. 413–430. DOI: 10.1007/3-540-48405-1_27. *(One citation on page 187.)*

[CP93] David Chaum and Torben Pryds Pedersen. "Wallet Databases with Observers". In: *Advances in Cryptology—CRYPTO*. Springer. 1993, pp. 89–105. DOI: 10.1007/3-540-48071-4_7. *(One citation on page 180.)*

[CPP06] Benoît Chevallier-Mames, Pascal Paillier, and David Pointcheval. "Encoding-Free ElGamal Encryption Without Random Oracles". In: *Public Key Cryptography (PKC)*. Springer. 2006, pp. 91–104. DOI: 10.1007/11745853_7. *(One citation on page 194.)*

[CPS+16a] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. "Improved OR Composition of Sigma-Protocols". In: *Theory of Cryptography*. Springer. 2016, pp. 112–141. DOI: 10.1007/978-3-662-49099-0_5. *(One citation on page 90.)*

[CPS+16b] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. "Online/Offline OR Composition of Sigma Protocols". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2016, pp. 63–92. DOI: 10.1007/978-3-662-49896-5_3. *(One citation on page 90.)*

[CS03a] Jan Camenisch and Victor Shoup. "Practical Verifiable Encryption and Decryption of Discrete Logarithms". In: *Advances in Cryptology—CRYPTO*. Springer. 2003, pp. 126–144. DOI: 10.1007/978-3-540-45146-4_8. *(One citation on page 183.)*

[CS03b] Ronald Cramer and Victor Shoup. "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack". In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226. DOI: 10.1137/S0097539702403773. *(Two citations on page 86.)*

[CS97] Jan Camenisch and Markus Stadler. "Efficient Group Signature Schemes for Large Groups". In: *Advances in Cryptology—CRYPTO*. Springer. 1997, pp. 410–424. DOI: 10.1007/BFb0052252. *(Eleven citations on pages 87, 90, 92, 179, 183, 184, 280, 373, 390, and 398.)*

[CSKH20] Camille Cobb, Lucy Simko, Tadayoshi Kohno, and Alexis Hiniker. "A Privacy-Focused Systematic Analysis of Online Status Indicators". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2020.3 (2020), pp. 384–403. DOI: 10.2478/popets-2020-0057. *(One citation on page 29.)*

[CT17] Yi-Ruei Chen and Wen-Guey Tzeng. “Group key management with efficient rekey mechanism: A Semi-Stateful approach for out-of-Synchronized members”. In: *Computer Communications* 98 (2017), pp. 31–42. DOI: 10.1016/j.comcom.2016.08.001.
(One citation on page 51.)

[CWF13] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. “Proactively Accountable Anonymous Messaging in Verdict”. In: *Security Symposium*. USENIX, 2013, pp. 147–162.
(One citation on page 37.)

[CYH05] Sherman S. M. Chow, Siu-Ming Yiu, and Lucas C. K. Hui. “Efficient Identity Based Ring Signature”. In: *Applied Cryptography and Network Security*. Springer. 2005, pp. 499–512. DOI: 10.1007/11496137_34. *(One citation on page 89.)*

[DABS17] Steve Dodier-Lazaro, Ruba Abu-Salma, Ingolf Becker, and M Angela Sasse. “From Paternalistic to User-Centred Security: Putting Users First with Value-Sensitive Design”. In: *ACM CHI2017 Workshop on Values in Computing*. 2017. *(One citation on page 17.)*

[Dam92] Ivan Damgård. “Towards Practical Public Key Systems Secure Against Chosen Ciphertext attacks”. In: *Advances in Cryptology—CRYPTO*. Springer. 1992, pp. 445–456. DOI: 10.1007/3-540-46766-1_36. *(One citation on page 272.)*

[Dav01] Don Davis. “Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML”. In: *Annual Technical Conference, General Track*. USENIX, 2001, pp. 65–78.
(One citation on page 30.)

[DB05] Ratna Dutta and Rana Barua. “Dynamic Group Key Agreement in Tree-Based Setting”. In: *Australasian Conference on Information Security and Privacy*. Springer. 2005, pp. 101–112. DOI: 10.1007/11506157_9. *(One citation on page 51.)*

[DB08] Ratna Dutta and Rana Barua. “Provably Secure Constant Round Contributory Group Key Agreement in Dynamic Setting”. In: *Transactions on Information Theory* 54.5 (2008), pp. 2007–2025. DOI: 10.1109/TIT.2008.920224. *(Two citations on pages 47 and 305.)*

[DBS04] Ratna Dutta, Rana Barua, and Palash Sarkar. “Provably Secure Authenticated Tree Based Group Key Agreement”. In: *International Conference on Information and Communications Security*. Vol. 4. Springer. 2004, pp. 92–104. DOI: 10.1007/978-3-540-30191-2_8.
(One citation on page 51.)

[DCE17] Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. “Indiscreet Logs: Diffie-Hellman Backdoors in TLS”. In: *Network and Distributed System Security Symposium (NDSS)*. Internet Society. 2017. DOI: 10.14722/ndss.2017.23006.
(Two citations on pages 205 and 206.)

[DDN00] Danny Dolev, Cynthia Dwork, and Moni Naor. “Nonmalleable Cryptography”. In: *SIAM Journal on Computing* 30.2 (2000), pp. 391–437. DOI: 10.1137/S0097539795291562.
(One citation on page 23.)

[DDO+16] Alexander De Luca, Sauvik Das, Martin Ortlieb, Iulia Ion, and Ben Laurie. "Expert and Non-Expert Attitudes towards (Secure) Instant Messaging". In: *Symposium on Usable Privacy and Security (SOUPS)*. 2016. *(One citation on page 14.)*

[DG19] Nir Drucker and Shay Gueron. "Fast Modular Squaring with AVX512IFMA". In: *Information Technology-New Generations (ITNG)*. Springer. 2019, pp. 3–8. DOI: 10.1007/978-3-030-14070-0_1. *(One citation on page 504.)*

[DGK05] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Secure Off-the-Record Messaging". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2005, pp. 81–89. DOI: 10.1145/1102199.1102216. *(One citation on page 107.)*

[DGK06] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Deniable Authentication and Key Exchange". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2006, pp. 400–409. DOI: 10.1145/1180405.1180454. *(Three citations on pages 23, 81, and 112.)*

[DH76] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography". In: *Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638. *(One citation on page 81.)*

[Din05] Roger Dingledine. *Tor security advisory: DH handshake flaw*. 2005. URL: http://archives.seul.org/or/announce/Aug-2005/msg00002.html (visited on 2021-04-24). *(One citation on page 102.)*

[DJMP11] Roy D'Souza, David Jao, Ilya Mironov, and Omkant Pandey. "Publicly Verifiable Secret Sharing for Cloud-based Key Management". In: *Progress in Cryptology—INDOCRYPT*. Springer. 2011, pp. 290–309. DOI: 10.1007/978-3-642-25578-6_21. *(One citation on page 183.)*

[DKNS04] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. "Anonymous Identification in Ad Hoc Groups". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2004, pp. 609–626. DOI: 10.1007/978-3-540-24676-3_36. *(Two citations on page 89.)*

[DKSW09] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. "Composability and On-Line Deniability of Authentication". In: *Theory of Cryptography*. Springer, 2009, pp. 146–162. DOI: 10.1007/978-3-642-00457-5_10. *(Nineteen citations on pages 23, 80, 81, 106, 112, 114, 116, 118–120, and 125.)*

[DLB07] Yvo Desmedt, Tanja Lange, and Mike Burmester. "Scalable Authenticated Tree Based Group Key Exchange for Ad-Hoc Groups". In: *Financial Cryptography and Data Security* (2007), pp. 104–118. DOI: 10.1007/978-3-540-77366-5_12. *(Two citations on page 49.)*

[DLKY04] Theodore Diament, Homin K. Lee, Angelos D. Keromytis, and Moti Yung. "The Dual Receiver Cryptosystem and Its Applications". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2004, pp. 330–343. DOI: 10.1145/1030083.1030128. *(Two citations on pages 84 and 85.)*

[DMC20] Jayati Dev, Pablo Moriano, and L. Jean Camp. “Lessons Learnt from Comparing WhatsApp Privacy Concerns Across Saudi and Indian Populations”. In: *Symposium on Usable Privacy and Security (SOUPS)*. USENIX. 2020, pp. 81–97. *(Two citations on pages 16 and 168.)*

[DMMK20] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. “Comprehensive Anonymity Trilemma: User Coordination is not enough”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2020.3 (2020), pp. 356–383. DOI: 10.2478/popets-2020-0056. *(One citation on page 28.)*

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: *Theory of Cryptography*. Springer, 2006, pp. 265–284. DOI: 10.1007/11681878_14. *(One citation on page 38.)*

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-Generation Onion Router”. In: *Security Symposium*. USENIX, 2004, pp. 147–162. *(One citation on page 38.)*

[DNDS19] Sergej Dechand, Alena Naiakshina, Anastasia Danilova, and Matthew Smith. “In Encryption We Don’t Trust: The Effect of End-To-End Encryption to the Masses on User Perception”. In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 401–415. DOI: 10.1109/eurosp.2019.00037. *(One citation on page 15.)*

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. “Concurrent Zero-Knowledge”. In: *Symposium on Theory of Computing*. ACM. 1998, pp. 409–418. DOI: 10.1145/276698.276853. *(One citation on page 23.)*

[DOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. “Authentication and Authenticated Key Exchanges”. In: *Designs, Codes and Cryptography* 2.2 (1992), pp. 107–125. DOI: 10.1007/BF00124891. *(One citation on page 30.)*

[DSB+18] Albesë Demjaha, Jonathan Spring, Ingolf Becker, Simon Parkin, and M Angela Sasse. “Metaphors considered harmful? An exploratory study of the effectiveness of functional metaphors for end-to-end encryption”. In: *Workshop on Usable Security (USEC)*. 2018. DOI: 10.14324/000.ds.10042529. *(One citation on page 16.)*

[Dwo15] Morris J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. NIST Federal Information Processing Standards FIPS 202. NIST, 2015, pp. 1–37. DOI: 10.6028/NIST.FIPS.202. *(One citation on page 408.)*

[EB16] Ame Elliott and Sara Brody. *Straight Talk: New Yorkers on Mobile Messaging and Implications for Privacy*. Tech. rep. Simply Secure, 2016. URL: https://simplysecure.org/what-we-do/NYC-study/ (visited on 2021-04-24). *(One citation on page 14.)*

[EHM17] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. “Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols”. In: *European Workshop on Usable Security (EuroUSEC)*. 2017. DOI: 10.14722/eurousec.2017.23016. *(Three citations on pages 2, 16, and 504.)*

[Ele17] Electric Coin Company. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. 2017. URL: https://electriccoin.co/blog/new-snark-curve/ (visited on 2021-04-24).
(Two citations on pages 274 and 288.)

[Ele19] Electric Coin Company. *What is Jubjub?* 2019. URL: https://z.cash/technology/jubjub/ (visited on 2021-04-24). *(Three citations on pages 274, 275, and 288.)*

[ElG85] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074. *(Six citations on pages 86, 87, 237, and 288.)*

[Ell07] Carl Ellison. *Ceremony Design and Analysis*. Tech. rep. 2007/399. Cryptology ePrint Archive, 2007. URL: https://eprint.iacr.org/2007/399 (visited on 2021-04-24).
(One citation on page 17.)

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code For Cryptographic Arithmetic — With Proofs, Without Compromises". In: *Symposium on Security and Privacy (S&P)*. IEEE. 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005. *(One citation on page 224.)*

[eQu15] eQualit.ie. *(N+1)SEC*. 2015. URL: https://learn.equalit.ie/wiki/Np1sec (visited on 2021-04-24). *(Two citations on pages 2 and 39.)*

[Fas18] Matthias Fassl. "Usable Authentication Ceremonies in Secure Instant Messaging". Diplom-Ingenieur thesis. TU Wien, 2018. URL: https://hdl.handle.net/20.500.12708/1817 (visited on 2021-04-24). *(One citation on page 17.)*

[FGHT17] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. "A Kilobit Hidden SNFS Discrete Logarithm Computation". In: *Advances in Cryptology—EUROCRYPT*. Springer, 2017, pp. 202–231. DOI: 10.1007/978-3-319-56620-7_8.
(Three citations on pages 206 and 210.)

[FGSW16] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3". In: *Symposium on Security and Privacy (S&P)*. IEEE. 2016, pp. 452–469. DOI: 10.1109/SP.2016.34.
(Three citations on pages 98, 100, and 138.)

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, and Uwe Sander. "Helping Johnny 2.0 to Encrypt His Facebook Conversations". In: *Symposium on Usable Privacy and Security (SOUPS)*. ACM. 2012. DOI: 10.1145/2335356.2335371. *(One citation on page 13.)*

[FJT13] Pierre-Alain Fouque, Antoine Joux, and Mehdi Tibouchi. "Injective Encodings to Elliptic Curves". In: *Information Security and Privacy*. Springer, 2013, pp. 203–218. DOI: 10.1007/978-3-642-39059-3_14. *(One citation on page 277.)*

[FKMV12] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. "On the Non-malleability of the Fiat-Shamir Transform". In: *Progress in Cryptology—INDOCRYPT*. Springer. 2012, pp. 60–79. DOI: 10.1007/978-3-642-34931-7_5.
(Five citations on pages 86, 89, and 93.)

[FL04] Denis Fomin and Yann Leboulanger. *Gajim, A fully-featured XMPP client*. 2004. URL: https://gajim.org/ (visited on 2021-04-24). *(Two citations on page 30.)*

[Flo62] Robert W. Floyd. "Algorithm 97: Shortest Path". In: *Communications of the ACM* 4.6 (1962), p. 345. DOI: 10.1145/367766.368168. *(One citation on page 178.)*

[FLS97] Alan Fekete, Nancy Lynch, and Alex Shvartsman. "Specifying and Using a Partitionable Group Communication Service". In: *Symposium on Principles of Distributed Computing (PODC)*. ACM. 1997, pp. 53–62. DOI: 10.1145/259380.259422. *(One citation on page 50.)*

[FM15] Marc Fischlin and Sogol Mazaheri. "Notions of Deniable Message Authentication". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM, 2015, pp. 55–64. DOI: 10.1145/2808138.2808143. *(One citation on page 112.)*

[FN93] Amos Fiat and Moni Naor. "Broadcast Encryption". In: *Advances in Cryptology—CRYPTO*. Springer. 1993, pp. 480–491. DOI: 10.1007/3-540-48329-2_40. *(One citation on page 84.)*

[FO11] Marc Fischlin and Cristina Onete. "Relaxed Security Notions for Signatures of Knowledge". In: *Applied Cryptography and Network Security*. Springer. 2011, pp. 309–326. DOI: 10.1007/978-3-642-21554-4_18. *(One citation on page 92.)*

[FS87] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology—CRYPTO*. Springer. 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12.
(Thirteen citations on pages 86, 90, 93, 179, 184, 194, 221, 246, 261, 376, 393, 397, and 400.)

[FSS+20] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. *Hashing to Elliptic Curves Draft 10*. Internet Draft. IETF, 2020. URL: https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-11 (visited on 2021-04-24).
(Two citations on pages 276 and 370.)

[Gab19] Ariel Gabizon. *On the security of the BCTV Pinocchio zk-SNARK variant*. Tech. rep. 2019/119. Cryptology ePrint Archive, 2019. URL: https://eprint.iacr.org/2019/119 (visited on 2021-04-24). *(One citation on page 272.)*

[GFF06] Shirley Gaw, Edward W. Felten, and Patricia Fernandez-Kelly. "Secrecy, Flagging, and Paranoia: Adoption Criteria in Encrypted E-Mail". In: *Conference on Human Factors in Computing Systems*. ACM. 2006, pp. 591–600. DOI: 10.1145/1124772.1124862.
(One citation on page 13.)

[GGHK21] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. *Stacking Sigmas: A Framework to Compose Σ-Protocols for Disjunctions*. Tech. rep. 2021/422. Cryptology ePrint Archive, 2021. URL: https://eprint.iacr.org/2021/422 (visited on 2021-04-24). *(One citation on page 506.)*

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: *Advances in Cryptology—EUROCRYPT*. Springer, 2013, pp. 626–645. DOI: 10.1007/978-3-642-38348-9_37. *(One citation on page 271.)*

[GHR20] Christine Geeng, Jevan Hutson, and Franziska Roesner. "Usable Sexurity: Studying People's Concerns and Strategies When Sexting". In: *Symposium on Usable Privacy and Security (SOUPS)*. USENIX. 2020, pp. 127–144. *(One citation on page 14.)*

[GJKR99] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems". In: *Advances in Cryptology—EUROCRYPT*. Springer. 1999, pp. 295–310. DOI: 10.1007/3-540-48910-X_21. *(Two citations on pages 76 and 78.)*

[GJY19] Qian Guo, Thomas Johansson, and Jing Yang. "A Novel CCA Attack Using Decryption Errors Against LAC". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2019, pp. 82–111. DOI: 10.1007/978-3-030-34578-5_4. *(One citation on page 104.)*

[GK19] Shay Gueron and Dusan Kostic. "Using the new VPMADD instructions for the new post quantum key encapsulation mechanism SIKE". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE. 2019, pp. 215–218. DOI: 10.1109/ARITH.2019.00050. *(One citation on page 504.)*

[GM05] Simson L. Garfinkel and Robert C. Miller. "Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express". In: *Symposium on Usable Privacy and Security (SOUPS)*. ACM. 2005, pp. 13–24. DOI: 10.1145/1073001.1073003. *(One citation on page 12.)*

[GM15] Matthew Green and Ian Miers. "Forward Secure Asynchronous Messaging from Puncturable Encryption". In: *Symposium on Security and Privacy (S&P)*. IEEE. 2015. DOI: 10.1109/SP.2015.26. *(Two citations on pages 31 and 68.)*

[GMP19] Steven Galbraith, Jake Massimo, and Kenneth G. Paterson. "Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation". In: *Public Key Cryptography (PKC)*. Springer. 2019, pp. 379–407. DOI: 10.1007/978-3-030-17259-6_13. *(One citation on page 209.)*

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks". In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308. DOI: 10.1137/0217017. *(One citation on page 101.)*

[GMY03] Juan A. Garay, Philip MacKenzie, and Ke Yang. "Strengthening Zero-Knowledge Protocols Using Signatures". In: *Advances in Cryptology—EUROCRYPT*. Vol. 2656. Springer. 2003, pp. 177–194. DOI: 10.1007/3-540-39200-9_11. *(One citation on page 118.)*

[Gol15] Mark Gollom. *Alain Philippon phone password case: Powers of border agents and police differ*. 2015. URL: http://www.cbc.ca/news/--1.2983841 (visited on 2021-04-24). *(One citation on page 108.)*

[Goo20] Google. *Protocol Buffers. Encoding: Base 128 Varints*. 2020. URL: https://developers.google.com/protocol-buffers/docs/encoding#varints (visited on 2021-04-24). *(One citation on page 464.)*

[Gor98] Daniel M. Gordon. "A Survey of Fast Exponentiation Methods". In: *Journal of Algorithms* 27.1 (1998), pp. 129–146. DOI: 10.1006/jagm.1997.0913. *(One citation on page 227.)*

[GPA19] Lachlan J. Gunn, Ricardo Vieitez Parra, and N. Asokan. "Circumventing Cryptographic Deniability with Remote Attestation". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2019.3 (2019), pp. 350–369. DOI: 10.2478/popets-2019-0051. *(One citation on page 77.)*

[Gro16] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology—EUROCRYPT*. Springer, 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11. *(Five citations on pages 271, 274, 275, 280, and 288.)*

[Gru17] Loren Grush. *A US-born NASA scientist was detained at the border until he unlocked his phone*. 2017. URL: https://www.theverge.com/2017/2/12/14583124/ (visited on 2021-04-24). *(One citation on page 108.)*

[Gue12] Shay Gueron. "Efficient Software Implementations of Modular Exponentiation". In: *Journal of Cryptographic Engineering* 2 (2012), pp. 31–43. DOI: 10.1007/s13389-012-0031-5. *(Two citations on pages 224 and 225.)*

[GUVC09] Ian Goldberg, Berkant Ustaoğlu, Matthew D. Van Gundy, and Hao Chen. "Multi-party Off-the-Record Messaging". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2009, pp. 358–368. DOI: 10.1145/1653662.1653705. *(Three citations on pages 2 and 35.)*

[GW11] Craig Gentry and Daniel Wichs. "Separating Succinct Non-Interactive Arguments From All Falsifiable Assumptions". In: *Symposium on Theory of Computing (STOC)*. ACM. 2011, pp. 99–108. DOI: 10.1145/1993636.1993651. *(One citation on page 273.)*

[GZH+18] Nina Gerber, Verena Zimmermann, Birgit Henhapl, Sinem Emeröz, and Melanie Volkamer. "Finally Johnny Can Encrypt. But Does This Make Him Feel More Secure?" In: *International Conference on Availability, Reliability and Security (ARES)*. ACM. 2018, pp. 1–10. DOI: 10.1145/3230833.3230859. *(One citation on page 15.)*

[HAK01] Fumitaka Hoshino, Masayuki Abe, and Tetsutaro Kobayashi. "Lenient/Strict Batch Verification in Several Groups". In: *Information Security*. Springer. 2001, pp. 81–94. DOI: 10.1007/3-540-45439-X_6. *(One citation on page 396.)*

[Hea12] Christopher C. D. Head. "Anonycaster: Simple, Efficient Anonymous Group Communication". 2012. URL: https://blogs.ubc.ca/computersecurity/files/2012/04/anonycaster.pdf (visited on 2021-04-24). *(One citation on page 37.)*

[HEM18] Harry Halpin, Ksenia Ermoshina, and Francesca Musiani. "Co-ordinating Developers and High-Risk Users of Privacy-Enhanced Secure Messaging Protocols". In: *Security Standardisation Research Conference (SSR)*. Springer. 2018, pp. 56–75. DOI: 10.1007/978-3-030-04762-7_4. *(Four citations on pages 2, 16, 165, and 504.)*

[Hen10] Ryan Henry. *Pippenger's Multiproduct and Multiexponentiation Algorithms. (Extended Version)*. Tech. rep. CACR 2010-26. University of Waterloo, 2010. URL: https://cacr.uwaterloo.ca/techreports/2010/cacr2010-26.pdf (visited on 2021-04-24). *(Two citations on pages 234 and 237.)*

[Hen14] Ryan Henry. "Efficient Zero-Knowledge Proofs and Applications". PhD thesis. University of Waterloo, 2014. URL: https://hdl.handle.net/10012/8621 (visited on 2021-04-24). *(Nine citations on pages 181, 182, 196, 244, 261, 379, 397, and 401.)*

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *Theory of Cryptography*. Springer, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. *(One citation on page 94.)*

[HKM15] Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. "Automated Analysis and Synthesis of Authenticated Encryption Schemes". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 84–95. DOI: 10.1145/2810103.2813636. *(One citation on page 86.)*

[HLZZ15] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. "Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis". In: *Symposium on Operating Systems Principles (SOSP)*. ACM. 2015, pp. 137–152. DOI: 10.1145/2815400.2815417. *(Two citations on pages 2 and 37.)*

[HMS03] Dennis Hofheinz, Jörn Müller-Quade, and Rainer Steinwandt. "Initiator-Resilient Universally Composable Key Exchange". In: *European Symposium on Research in Computer Security (ESORICS)*. Springer. 2003, pp. 61–84. DOI: 10.1007/978-3-540-39650-5_4. *(Nine citations on pages 81, 118–120, and 124.)*

[HN04] Johan Håstad and Mats Näslund. "The Security of all RSA and Discrete Log Bits". In: *Journal of the ACM* 51.2 (2004), pp. 187–230. DOI: 10.1145/972639.972642. *(One citation on page 264.)*

[Hod20] Rae Hodge. *Zoom security issues: Zoom buys security company, aims for end-to-end encryption*. 2020. URL: https://www.cnet.com/news/zoom-security-issues-zoom-buys-security-company-aims-for-end-to-end-encryption/ (visited on 2021-04-24).
(One citation on page 1.)

[HT98] Satoshi Hada and Toshiaki Tanaka. "On the Existence of 3-Round Zero-Knowledge Protocols". In: *Advances in Cryptology—CRYPTO*. Springer. 1998, pp. 408–423. DOI: 10.1007/BFb0055744.
(One citation on page 272.)

[ITW82] Ingemar Ingemarsson, Donald Tang, and C. Wong. "A Conference Key Distribution System". In: *Transactions on Information Theory* 28.5 (1982), pp. 714–720. DOI: 10.1109/TIT.1982.1056542.
(One citation on page 43.)

[JKT07] Stanisław Jarecki, Jihye Kim, and Gene Tsudik. "Robust Group Key Agreement Using Short Broadcasts". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2007, pp. 411–420. DOI: 10.1145/1315245.1315296.
(One citation on page 47.)

[Joh14] Johns Hopkins University. *The Johns Hopkins Foreign Affairs Symposium Presents: The Price of Privacy: Re-Evaluating the NSA*. 2014. URL: https://www.youtube.com/watch?v=kV2HDM86XgI (visited on 2021-04-24).
(One citation on page 28.)

[Jou00] Antoine Joux. "A One Round Protocol for Tripartite Diffie–Hellman". In: *International Algorithmic Number Theory Symposium (ANTS)*. Springer. 2000, pp. 385–393. DOI: 10.1007/10722028_23.
(Two citations on pages 42 and 51.)

[JS08] Shaoquan Jiang and Reihaneh Safavi-Naini. "An Efficient Deniable Key Exchange Protocol". In: *Financial Cryptography and Data Security*. Springer, 2008. DOI: 10.1007/978-3-540-85230-8_4.
(One citation on page 23.)

[Kad16] George Kadianakis. *flute Specification*. 2016. URL: https://github.com/asn-d6/flute (visited on 2021-04-24).
(Two citations on pages 2 and 38.)

[Kat03] Jonathan Katz. "Efficient and Non-Malleable Proofs of Plaintext Knowledge and Applications". In: *Advances in Cryptology—EUROCRYPT*. Springer, 2003, pp. 211–228. DOI: 10.1007/3-540-39200-9_13.
(One citation on page 23.)

[KB16] Taechan Kim and Razvan Barbulescu. "Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case". In: *Advances in Cryptology—CRYPTO*. Springer. 2016, pp. 543–571. DOI: 10.1007/978-3-662-53018-4_20.
(Two citations on pages 84 and 272.)

[KCDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. "Atom: Horizontally Scaling Strong Anonymity". In: *Symposium on Operating Systems Principles (SOSP)* 21.46 (2017), p. 72. DOI: 10.1145/3132747.3132755.
(One citation on page 37.)

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash*. NIST Special Publication SP 800-185. NIST, 2016, pp. 1–32. DOI: 10.6028/NIST.SP.800-185.
(Five citations on pages 105, 408, 479, and 487.)

[KG20] Chelsea Komlo and Ian Goldberg. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. Tech. rep. 2020/852. Cryptology ePrint Archive, 2020. URL: https://eprint.iacr.org/2020/852 (visited on 2021-04-24). *(One citation on page 307.)*

[KLL04] Hyun-Jeong Kim, Su-Mi Lee, and Dong Hoon Lee. "Constant-Round Authenticated Group Key Exchange for Dynamic Groups". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2004, pp. 245–259. DOI: 10.1007/978-3-540-30539-2_18. *(Five citations on pages 46, 294, 299, 305, and 322.)*

[KM15] Neal Koblitz and Alfred J. Menezes. "The random oracle model: a twenty-year retrospective". In: *Designs, Codes and Cryptography* 77.2-3 (2015), pp. 587–610. DOI: 10.1007/s10623-015-0094-2. *(One citation on page 80.)*

[KMS05] Chisato Konoma, Masahiro Mambo, and Hiroki Shizuya. "The Computational Difficulty of Solving Cryptographic Primitive Problems Related to the Discrete Logarithm Problem". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 88.1 (2005), pp. 81–88. DOI: 10.1093/ietfec/e88-a.1.81. *(One citation on page 183.)*

[Knu97] Donald E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*. 3rd ed. 4 vols. Vol. 2. Addison-Wesley, 1997. ISBN: 0-201-89684-2. URL: https://www-cs-faculty.stanford.edu/~knuth/taocp.html (visited on 2021-04-24). *(Two citations on pages 227 and 234.)*

[KPT00] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2000, pp. 235–244. DOI: 10.1145/352600.352638. *(Two citations on pages 50 and 203.)*

[KPT02] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Communication-Efficient Group Key Agreement". In: *Trusted Information*. Springer, 2002, pp. 229–244. DOI: 10.1007/0-306-46998-7_16. *(One citation on page 43.)*

[KPT04] Yongdae Kim, Adrian Perrig, and Gene Tsudik. "Tree-Based Group Key Agreement". In: *Transactions on Information and System Security (TISSEC)* 7.1 (2004), pp. 60–96. DOI: 10.1145/984334.984337. *(Three citations on pages 51 and 203.)*

[Kra03] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE protocols". In: *Advances in Cryptology—CRYPTO*. Springer, 2003, pp. 400–425. DOI: 10.1007/978-3-540-45146-4_24. *(Four citations on pages 23, 31, 100, and 106.)*

[Kra10] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *Advances in Cryptology—CRYPTO*. Springer. 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34. *(One citation on page 219.)*

[Kra96] Hugo Krawczyk. “SKEME: A Versatile Secure Key Exchange Mechanism for Internet”. In: *Network and Distributed System Security Symposium (NDSS)*. IEEE. 1996, pp. 114–127. DOI: 10.1109/NDSS.1996.492418. *(One citation on page 23.)*

[KTN04] Hiroaki Kikuchi, Minako Tada, and Shohachiro Nakanishi. “Secure Instant Messaging Protocol Preserving Confidentiality against Administrator”. In: *International Conference on Advanced Information Networking and Applications*. IEEE. 2004, pp. 27–30. DOI: 10.1109/AINA.2004.1283749. *(Two citations on pages 2 and 33.)*

[KV19] Kris Kwiatkowski and Luke Valenta. *The TLS Post-Quantum Experiment*. 2019. URL: https://blog.cloudflare.com/the-tls-post-quantum-experiment/ (visited on 2021-04-24).

(Two citations on pages 83 and 94.)

[KY03] Jonathan Katz and Moti Yung. “Scalable Protocols for Authenticated Group Key Exchange”. In: *Advances in Cryptology—CRYPTO*. Vol. 3. Springer. 2003, pp. 110–125. DOI: 10.1007/978-3-540-45146-4_7. *(Four citations on pages 41, 47, 49, and 305.)*

[Lan16] Adam Langley. *Intent to Implement and Ship: CECPQ1 for TLS*. 2016. URL: https://groups.google.com/a/chromium.org/g/security-dev/c/DS9pp2U0SAc (visited on 2021-04-24). *(Two citations on pages 83 and 94.)*

[Lew18] Sarah Jamie Lewis. *Cwtch: Privacy Preserving Infrastructure for Asynchronous, Decentralized, Multi-Party and Metadata Resistant Applications*. Tech. rep. Open Privacy Research Society, 2018. URL: https://cwtch.im/cwtch.pdf (visited on 2021-04-24).

(One citation on page 38.)

[Ley02] Paul Leyland. *The comp.security.pgp FAQ. Q: Can a public key be forged?* Ed. by Wouter Slegers. 2002. URL: http://www.pgp.net/pgpnet/pgp-faq/#KEY-PUBLIC-KEY-FORGERY (visited on 2021-04-24). *(One citation on page 361.)*

[Lin03] Yehuda Lindell. “General Composition and Universal Composability in Secure Multi-Party Computation”. In: *Symposium on Foundations of Computer Science*. IEEE. 2003, pp. 394–403. DOI: 10.1109/SFCS.2003.1238213. *(One citation on page 116.)*

[LK16] Ximin Luo and Guy Kloss. *mpENC: Multi-Party Encrypted Messaging Protocol design document*. Tech. rep. 1606.04598. arXiv, 2016. URL: http://arxiv.org/abs/1606.04598 (visited on 2021-04-24). *(Two citations on pages 2 and 39.)*

[LKKR03] Sangwon Lee, Yongdae Kim, Kwangjo Kim, and DaeHyun Ryu. “An Efficient Tree-based Group Key Agreement using Bilinear Map”. In: *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer. 2003, pp. 357–371. DOI: 10.1007/978-3-540-45203-4_28. *(One citation on page 51.)*

[LL97] Chae Hoon Lim and Pil Joong Lee. “A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup”. In: *Advances in Cryptology—CRYPTO*. Springer, 1997, pp. 249–263. DOI: 10.1007/BFb0052240. *(One citation on page 102.)*

[LLM07] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. “Stronger Security of Authenticated Key Exchange”. In: *Provable Security (ProvSec)*. Springer. 2007, pp. 1–16. DOI: 10.1007/978-3-540-75670-5_1. *(Two citations on pages 39 and 177.)*

[LSL16] Dong Lin, Micah Sherr, and Boon Thau Loo. “Scalable and Anonymous Group Communication with MTor”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2016.2 (2016), pp. 22–39. DOI: 10.1515/popets-2016-0003. *(One citation on page 38.)*

[Lun18] Joshua Lund. *Signal partners with Microsoft to bring end-to-end encryption to Skype*. 2018. URL: https://signal.org/blog/skype-partnership/ (visited on 2021-04-24). *(One citation on page 2.)*

[Luo14] Ximin Luo. *Causal ordering*. 2014. URL: https://github.com/infinity0/msg-notes/blob/master/causal/index.rst (visited on 2021-04-24). *(One citation on page 34.)*

[LVH13] Hong Liu, Eugene Y. Vasserman, and Nicholas Hopper. “Improved Group Off-the-Record Messaging”. In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2013, pp. 249–254. DOI: 10.1145/2517840.2517867. *(Three citations on pages 2, 36, and 48.)*

[LWW04] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. “Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups”. In: *Australasian Conference on Information Security and Privacy*. Springer. 2004, pp. 325–335. DOI: 10.1007/978-3-540-27800-9_28. *(One citation on page 89.)*

[LX13] Chengbang Li and Chunxiang Xu. “Scalable group key exchange protocol with provable security”. In: *COMPEL* 32.2 (2013), pp. 612–619. DOI: 10.1108/03321641311296990. *(One citation on page 48.)*

[Mad15] Marry Madden. *Americans’ Attitudes About Privacy, Security and Surveillance*. 2015. URL: https://www.pewresearch.org/internet/2015/05/20/americans-attitudes-about-privacy-security-and-surveillance/ (visited on 2021-04-24). *(One citation on page 1.)*

[Mar13] Moxie Marlinspike. *Simplifying OTR deniability*. 2013. URL: https://signal.org/blog/simplifying-otr-deniability/ (visited on 2021-04-24). *(Four citations on pages 23, 81, 100, and 106.)*

[Mar14a] Moxie Marlinspike. *Open Whisper Systems partners with WhatsApp to provide end-to-end encryption*. 2014. URL: https://signal.org/blog/whatsapp/ (visited on 2021-04-24). *(One citation on page 1.)*

[Mar14b] Moxie Marlinspike. *Private Group Messaging*. 2014. URL: https://signal.org/blog/private-groups/ (visited on 2021-04-24). *(One citation on page 36.)*

[Mar15] Marcel Martin. *Primo*. 2015. URL: https://www.ellipsa.eu/ (visited on 2021-04-24). *(One citation on page 210.)*

[Mar16] Moxie Marlinspike. *Facebook Messenger deploys Signal Protocol for end to end encryption*. 2016. URL: https://signal.org/blog/facebook-messenger/ (visited on 2021-04-24).

(One citation on page 1.)

[Mat15] Mattermost. *Mattermost: High Trust Messaging for the Enterprise*. 2015. URL: https://mattermost.com/ (visited on 2021-04-24). *(One citation on page 19.)*

[MBZ12] Vinnie Moscaritolo, Gary Belvin, and Phil Zimmermann. *Silent Circle Instant Messaging Protocol Protocol Specification*. Silent Circle, 2012. *(One citation on page 32.)*

[McL20] Michael McLoughlin. *addchain. Cryptographic Addition Chain Generation in Go*. 2020. URL: https://github.com/mmcloughlin/addchain (visited on 2021-04-24).

(One citation on page 232.)

[Mil76] Gary L. Miller. "Riemann's Hypothesis and Tests for Primality". In: *Journal of Computer and System Sciences* 13.3 (1976), pp. 300–317. DOI: 10.1016/S0022-0000(76)80043-8.

(Two citations on pages 208 and 209.)

[MO06] Mohammad Mannan and Paul C. van Oorschot. "A Protocol for Secure Public Instant Messaging". In: *Financial Cryptography and Data Security*. Springer, 2006, pp. 20–35. DOI: 10.1007/11889663_2. *(One citation on page 31.)*

[Mon85] Peter L. Montgomery. "Modular Multiplication Without Trial Division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. DOI: 10.1090/S0025-5718-1985-0777282-X. *(One citation on page 223.)*

[Moo12] Andrew Moon. *Implementations of a fast Elliptic-curve Digital Signature Algorithm*. 2012. URL: https://github.com/floodyberry/ed25519-donna (visited on 2021-04-24).

(One citation on page 104.)

[Mor98] François Morain. "Primality proving using elliptic curves: An update". In: *Algorithmic Number Theory*. Springer. 1998, pp. 111–127. DOI: 10.1007/BFb0054855.

(One citation on page 209.)

[MOV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. With a forew. by Ronald L. Rivest. DOI (EBOOK): 10.1201/9780429466335. CRC Press, 1997. ISBN: 978-0-84-938523-0. *(Two citations on pages 227 and 232.)*

[MP15] Gregory Maxwell and Andrew Poelstra. *Borromean Ring Signatures*. Tech. rep. Blockstream, 2015. URL: https://github.com/Blockstream/borromean_paper (visited on 2021-04-24). *(Three citations on pages 400, 404, and 506.)*

[MP16] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. 2016. URL: https://signal.org/docs/specifications/x3dh/ (visited on 2021-04-24).

(Eight citations on pages 23, 32, 74, 76, 100, 101, and 106.)

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. "Simple Schnorr Multi-Signatures with Applications to Bitcoin". In: *Designs, Codes and Cryptography* 87.9 (2019), pp. 2139–2164. DOI: 10.1007/s10623-019-00608-x. *(One citation on page 307.)*

[MW96] Ueli M. Maurer and Stefan Wolf. "Diffie-Hellman Oracles". In: *Advances in Cryptology—CRYPTO*. Springer. 1996, pp. 268–282. DOI: 10.1007/3-540-68697-5_21. *(One citation on page 190.)*

[MZ17] Fermi Ma and Mark Zhandry. *Encryptor Combiners: A Unified Approach to Multiparty NIKE, (H)IBE, and Broadcast Encryption*. Tech. rep. 2017/152. Cryptology ePrint Archive, 2017. URL: https://eprint.iacr.org/2017/152 (visited on 2021-04-24). *(One citation on page 42.)*

[NBMV99] Khanh Quoc Nguyen, Feng Bao, Yi Mu, and Vijay Varadharajan. "Zero-Knowledge Proofs of Possession of Digital Signatures and Its Applications". In: *Information and Communications Security*. Springer. 1999, pp. 103–118. DOI: 10.1007/978-3-540-47942-0_9. *(One citation on page 187.)*

[NHS99] Toru Nakanishi, Nobuaki Haruna, and Yuji Sugiyama. "Unlinkable Electronic Coupon Protocol with Anonymity Control". In: *Information Security (ISW)*. Springer. 1999, pp. 37–46. DOI: 10.1007/3-540-47790-X_4. *(Two citations on pages 183 and 185.)*

[NL18] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. IETF, 2018, pp. 1–45. URL: https://tools.ietf.org/html/rfc8439 (visited on 2021-04-24). *(One citation on page 464.)*

[NRS20] Jonas Nick, Tim Ruffing, and Yannick Seurin. *MuSig2: Simple Two-Round Schnorr Multi-Signatures*. Tech. rep. 2020/1261. Cryptology ePrint Archive, 2020. URL: https://eprint.iacr.org/2020/1261 (visited on 2021-04-24). *(One citation on page 307.)*

[NY90] Moni Naor and Moti Yung. "Public-key Cryptosystems Provably Secure against Chosen Ciphertext Attacks". In: *Symposium on Theory of Computing*. ACM. 1990, pp. 427–437. DOI: 10.1145/100216.100273. *(One citation on page 86.)*

[OGG13] Erdinc Ozturk, James Guilford, and Vinodh Gopal. *Large Integer Squaring on Intel® Architecture Processors*. White paper 328569-001. Intel, 2013. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf (visited on 2021-04-24). *(Two citations on pages 225 and 226.)*

[OGGF12] Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghali. *New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors*. White paper 327831-001. Intel, 2012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf (visited on 2021-04-24). *(Two citations on pages 224 and 227.)*

[Ope13] Open Whisper Systems. *Signal*. 2013. URL: https://signal.org/ (visited on 2021-04-24).
(Nine citations on pages 2, 17, 23, 32, 36, 74, 105, 496, and 502.)

[OTR14] OTR.im. *OTR capable clients*. 2014. URL: https://otr.im/clients.html (visited on 2021-04-24). *(Two citations on pages 2 and 19.)*

[OTR16] OTR Development Team. *Off-the-Record Messaging Protocol version 3*. 2016. URL: https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html (visited on 2021-04-24).
(Three citations on pages 19, 74, and 75.)

[OTR17] OTRv4 Development Team. *Off-the-Record Messaging Protocol version 4*. 2017. URL: https://github.com/otrv4/otrv4 (visited on 2021-04-24). *(Two citations on pages 22 and 74.)*

[PBD07] Kun Peng, Colin Boyd, and Ed Dawson. "Batch Zero-Knowledge Proof and Verification and Its Applications". In: *ACM Transactions on Information and System Security (TISSEC)* 10.2 (2007), p. 6. DOI: 10.1145/1237500.1237502.
(Four citations on pages 181, 182, 376, and 397.)

[Ped91] Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Advances in Cryptology—CRYPTO*. Springer. 1991, pp. 129–140. DOI: 10.1007/3-540-46766-1_9. *(One citation on page 274.)*

[PH78] Stephen C. Pohlig and Martin E. Hellman. "An Improved Algorithm for Computing Logarithms over $GF(p)$ and Its Cryptographic Significance". In: *IEEE Transactions on Information Theory* 24.1 (1978), pp. 106–110. DOI: 10.1109/TIT.1978.1055817.
(One citation on page 206.)

[Pip76] Nicholas Pippenger. "On the Evaluation of Powers and Related Problems. (Preliminary Version)". In: *Symposium on Foundations of Computer Science*. IEEE. 1976, pp. 258–263. DOI: 10.1109/SFCS.1976.21. *(Three citations on pages 234, 237, and 238.)*

[Pip78] Nicholas Pippenger. "The Minimum Number of Edges in Graphs with Prescribed Paths". In: *Mathematical Systems Theory* 12 (1978), pp. 325–346. DOI: 10.1007/BF01776581.
(One citation on page 234.)

[Pip80] Nicholas Pippenger. "On the Evaluation of Powers and Monomials". In: *Journal on Computing* 9.2 (1980), pp. 230–250. DOI: 10.1137/0209022.
(One citation on page 234.)

[PM16] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. 2016. URL: https://signal.org/docs/specifications/doubleratchet/ (visited on 2021-04-24).
(Three citations on pages 32 and 74.)

[Poc1914] Henry C. Pocklington. "The determination of the prime or composite nature of large numbers by Fermat's theorem". In: *Proceedings of the Cambridge Philosophical Society* 18 (1914), pp. 29–30. ARK: ark:/13960/t8pc3kb33. *(One citation on page 209.)*

[Pol00] J. M. Pollard. “Kangaroos, Monopoly and Discrete Logarithms”. In: *Journal of Cryptology* 13.4 (2000), pp. 437–447. DOI: 10.1007/s001450010010. *(One citation on page 206.)*

[Pol75] J. M. Pollard. “A Monte Carlo Method for Factorization”. In: *BIT Numerical Mathematics* 15.3 (1975), pp. 331–334. DOI: 10.1007/BF01933667. *(One citation on page 205.)*

[Pom84] Carl Pomerance. “Are There Counter-Examples to the Baillie-PSW Primality Test?” 1984. URL: https://math.dartmouth.edu/~carlp/dopo.pdf (visited on 2021-04-24). *(One citation on page 209.)*

[PPS12] Duong Hieu Phan, David Pointcheval, and Mario Strefler. “Decentralized Dynamic Broadcast Encryption”. In: *International Conference on Security and Cryptography for Networks (SCN)*. Vol. 12. Springer. 2012, pp. 166–183. DOI: 10.1007/978-3-642-32928-9_10. *(One citation on page 52.)*

[Pri82] Paul Pritchard. “Explaining the Wheel Sieve”. In: *Acta Informatica* 17.4 (1982), pp. 477–485. DOI: 10.1007/BF00264164. *(One citation on page 213.)*

[Pri83] Paul Pritchard. “Fast Compact Prime Number Sieves (among Others)”. In: *Journal of Algorithms* 4.4 (1983), pp. 332–344. DOI: 10.1016/0196-6774(83)90014-7. *(One citation on page 213.)*

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. “SoK: Game-Based Security Models for Group Key Exchange”. In: *Topics in Cryptology—CT-RSA*. Springer. 2021, pp. 148–176. DOI: 10.1007/978-3-030-75539-3_7. *(One citation on page 506.)*

[PSW80] Carl Pomerance, John L. Selfridge, and Samuel S. Wagstaff Jr. “The Pseudoprimes to $25 \cdot 10^9$”. In: *Mathematics of Computation* 35.151 (1980), pp. 1003–1026. DOI: 10.1090/S0025-5718-1980-0572872-7. *(Two citations on pages 209 and 212.)*

[RAD+19] Scott Ruoti, Jeff Andersen, Luke Dickinson, Scott Heidbrink, Tyler Monson, Mark O’Neill, Ken Reese, Brad Spendlove, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. “A Usability Study of Four Secure Email Tools Using Paired Participants”. In: *ACM Transactions on Privacy and Security (TOPS)* 22.2 (2019), pp. 1–33. DOI: 10.1145/3313761. *(One citation on page 14.)*

[RAH+16a] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O’Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. ““We’re on the Same Page”: A Usability Study of Secure Email Using Pairs of Novice Users”. In: *Conference on Human Factors in Computing Systems*. ACM. 2016, pp. 4298–4308. DOI: 10.1145/2858036.2858400. *(One citation on page 14.)*

[RAH+16b] Scott Ruoti, Jeff Andersen, Travis Hendershot, Daniel Zappala, and Kent Seamons. “Private Webmail 2.0: Simple and Easy-to-Use Secure Email”. In: *Symposium on User Interface Software and Technology (UIST)*. ACM. 2016, pp. 461–472. DOI: 10.1145/2984511.2984580. *(One citation on page 14.)*

[RAZS15] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. *Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client*. Tech. rep. 1510.08555. arXiv, 2015. URL: https://arxiv.org/abs/1510.08555 (visited on 2021-04-24).
(One citation on page 14.)

[RGK14] Franziska Roesner, Brian T. Gill, and Tadayoshi Kohno. "Sex, Lies, or Kittens? Investigating the Use of Snapchat's Self-Destructing Messages". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 64–76. DOI: 10.1007/978-3-662-45472-5_5.
(One citation on page 14.)

[Ric14] Ricochet Project. *Anonymous metadata-resistant instant messaging that just works*. 2014. URL: https://github.com/ricochet-im/ricochet (visited on 2021-04-24).
(Three citations on pages 2, 30, and 38.)

[RKAG07] Joel Reardon, Alan Kligman, Brian Agala, and Ian Goldberg. *KleeQ: Asynchronous Key Management for Dynamic Ad-Hoc Networks*. Tech. rep. CACR 2007-03. University of Waterloo, 2007. URL: https://cacr.uwaterloo.ca/techreports/2007/cacr2007-03.pdf (visited on 2021-04-24).
(Two citations on pages 2 and 35.)

[RKB+13] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy van der Horst, and Kent Seamons. "Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes". In: *Symposium on Usable Privacy and Security (SOUPS)*. ACM. 2013. DOI: 10.1145/2501604.2501609.
(One citation on page 13.)

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. "More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema". In: *European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036.
(One citation on page 3.)

[Rog02] Phillip Rogaway. "Authenticated-Encryption with Associated-Data". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2002, pp. 98–107. DOI: 10.1145/586110.586125.
(One citation on page 85.)

[Rog15] Phillip Rogaway. *The Moral Character of Cryptographic Work*. Tech. rep. 2015/1162. Cryptology ePrint Archive, 2015. URL: https://eprint.iacr.org/2015/1162 (visited on 2021-04-24).
(One citation on page 1.)

[Ros16] Luke Rosiak. *Here's Cryptographic Proof That Donna Brazile Is Wrong, WikiLeaks Emails Are Real*. 2016. URL: https://dailycaller.com/2016/10/21/heres-cryptographic-proof-that-donna-brazile-is-wrong-wikileaks-emails-are-real/ (visited on 2021-04-24).
(One citation on page 72.)

[RST+14] Michael Rogers, Eleanor Saitta, Bernard Tyers, Torsten Grote, Jack Grigg, and Ernir Erlingsson. *Briar*. 2014. URL: https://briarproject.org/ (visited on 2021-04-24).
(One citation on page 2.)

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2001, pp. 552–565. DOI: 10.1007/3-540-45682-1_32. *(One citation on page 84.)*

[RT10] Blake Ramsdell and Sean Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751. IETF, 2010, pp. 1–45. URL: https://tools.ietf.org/html/rfc5751 (visited on 2021-04-24). *(One citation on page 30.)*

[RVR14] Karen Renaud, Melanie Volkamer, and Arne Renkema-Padmos. "Why Doesn't Jane Protect Her Privacy?" In: *Privacy Enhancing Technologies Symposium (PETS)*. Springer. 2014, pp. 244–262. DOI: 10.1007/978-3-319-08506-7_13. *(Three citations on pages 13 and 15.)*

[Sai11] Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. RFC 6120. IETF, 2011, pp. 1–211. URL: https://tools.ietf.org/html/rfc6120 (visited on 2021-04-24). *(One citation on page 30.)*

[SBKH06] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J. Hyland. "Why Johnny Still Can't Encrypt: Evaluating the Usability of Email Encryption Software". In: *Symposium On Usable Privacy and Security (SOUPS)*. 2006, pp. 3–4. *(Two citations on pages 12 and 13.)*

[Sch15] Sven Schäge. "TOPAS: 2-Pass Key Exchange with Full Perfect Forward Secrecy and Optimal Communication Complexity". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 1224–1235. DOI: 10.1145/2810103.2813683. *(One citation on page 23.)*

[Sch91] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards". In: *Journal of Cryptology* 4.3 (1991), pp. 161–174. DOI: 10.1007/BF00196725. *(Fourteen citations on pages 90, 179, 182, 185, 188, 203, 270, 273, 282, 287, 312, 376, 394, and 410.)*

[Sch99] Berry Schoenmakers. "A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting". In: *Advances in Cryptology—CRYPTO*. Springer. 1999, pp. 148–164. DOI: 10.1007/3-540-48405-1_10. *(One citation on page 183.)*

[SH19] Michael Schliep and Nicholas Hopper. "End-to-End Secure Mobile Group Messaging with Conversation Integrity and Deniability". In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2019, pp. 55–73. DOI: 10.1145/3338498.3358644. *(Two citations on pages 2 and 39.)*

[Sha71] Daniel Shanks. "Class number, a theory of factorization and genera". In: *Proceedings of Symposia in Pure Math* 20 (1971). DOI (VOLUME): 10.1090/pspum/020, pp. 415–440. *(One citation on page 205.)*

[Sho04] Victor Shoup. *Sequences of Games: A Tool for Taming Complexity in Security Proofs*. Tech. rep. 2004/332. Cryptology ePrint Archive, 2004. URL: https://eprint.iacr.org/2004/332 (visited on 2021-04-24). *(One citation on page 260.)*

[SILC00] SILC Project. *SILC – Secure Internet Live Conferencing*. 2000. URL: http://silcnet.org/ (visited on 2021-04-24). *(One citation on page 33.)*

[Sim18] Sean Simpson. *Four in Ten (39%) Canadians Changed Their Social Media Behaviour (28%) or Stopped Using some Platforms (11%) Over Data Privacy Concerns*. 2018. URL: https://www.ipsos.com/en-ca/news-polls/Global-News-Data-Privacy-and-Social-Media-Poll-April-2018 (visited on 2021-04-24). *(One citation on page 1.)*

[SKH17] Michael Schliep, Ian Kariniemi, and Nicholas Hopper. "Is Bob Sending Mixed Signals?" In: *Workshop on Privacy in the Electronic Society (WPES)*. ACM. 2017, pp. 31–40. DOI: 10.1145/3139550.3139568. *(One citation on page 3.)*

[SMW07] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 17.9 (2007), pp. 1103–1120. DOI: 10.1109/TCSVT.2007.905532. *(One citation on page 500.)*

[SP06] Krishna Sampigethaya and Radha Poovendran. "A Survey on Mix Networks and Their Secure Applications". In: *Proceedings of the IEEE* 94.12 (2006), pp. 2142–2181. DOI: 10.1109/JPROC.2006.889687. *(One citation on page 37.)*

[SPG21] Michael A. Specter, Sunoo Park, and Matthew Green. "KeyForge: Non-Attributable Email from Forward-Forgeable Signatures". In: *Security Symposium*. USENIX, 2021. *(One citation on page 72.)*

[SSDW90] David G. Steer, Leo Strawczynski, Whitfield Diffie, and M. Wiener. "A Secure Audio Teleconference System". In: *Advances in Cryptology—CRYPTO*. Springer. 1990, pp. 520–528. DOI: 10.1007/0-387-34799-2_37. *(One citation on page 43.)*

[Sta96] Markus Stadler. "Publicly Verifiable Secret Sharing". In: *Advances in Cryptology—EUROCRYPT*. Springer. 1996, pp. 190–199. DOI: 10.1007/3-540-68339-9_17. *(Four citations on pages 183 and 185.)*

[Str64] Ernst G. Straus. "Addition Chains of Vectors". Advanced Problems: partial solution to problem 5125. In: *The American Mathematical Monthly* 71.7 (1964), pp. 806–808. DOI: 10.1080/00029890.1964.11992322. *(Two citations on pages 237 and 240.)*

[STW96] Michael Steiner, Gene Tsudik, and Michael Waidner. "Diffie-Hellman Key Distribution Extended to Group Communication". In: *Conference on Computer and Communications Security (CCS)*. ACM. 1996, pp. 31–37. DOI: 10.1145/238168.238182. *(One citation on page 44.)*

[STW98] Michael Steiner, Gene Tsudik, and Michael Waidner. "CLIQUES: A New Approach to Group Key Agreement". In: *International Conference on Distributed Computing Systems*. IEEE. 1998, pp. 380–387. DOI: 10.1109/ICDCS.1998.679745. *(One citation on page 44.)*

[SVH18] Michael Schliep, Eugene Vasserman, and Nicholas Hopper. "Consistent Synchronous Group Off-The-Record Messaging with SYM-GOTR". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2018.3 (2018), pp. 181–202. DOI: 10.1515/popets-2018-0027. *(Three citations on pages 2 and 36.)*

[SW07] Hovav Shacham and Brent Waters. “Efficient Ring Signatures without Random Oracles”. In: *Public Key Cryptography*. Springer, 2007, pp. 166–180. DOI: 10.1007/978-3-540-71677-8_12. *(One citation on page 90.)*

[SWB19] Josh Swihart, Benjamin Winston, and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. 2019. URL: https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/ (visited on 2021-04-24). *(One citation on page 272.)*

[SWZ16] John M. Schanck, William Whyte, and Zhenfei Zhang. “Circuit-extension handshakes for Tor achieving forward secrecy in a quantum world”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2016.4 (2016), pp. 219–236. DOI: 10.1515/popets-2016-0037. *(One citation on page 81.)*

[SYG08] Ryan Stedman, Kayo Yoshida, and Ian Goldberg. “A User Study of Off-the-Record Messaging”. In: *Symposium on Usable Privacy and Security (SOUPS)*. ACM. 2008, pp. 95–104. DOI: 10.1145/1408664.1408678. *(One citation on page 33.)*

[Tel14] Telegram. *Telegram Messenger*. 2014. URL: https://telegram.org/ (visited on 2021-04-24). *(Two citations on pages 2 and 15.)*

[TGL+17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. “Stadium: A Distributed Metadata-Private Messaging System”. In: *Symposium on Operating Systems Principles (SOSP)*. ACM. 2017, pp. 423–440. DOI: 10.1145/3132747.3132783. *(One citation on page 38.)*

[Thu73] Edward G. Thurber. “On Addition Chains $l(mn) \leq l(n) - b$ and Lower Bounds for $c(r)$”. In: *Duke Mathematical Journal* 40 (1973), pp. 907–913. DOI: 10.1215/S0012-7094-73-04085-4. *(One citation on page 228.)*

[Tsu92] Gene Tsudik. “Message Authentication with One-Way Hash Functions”. In: *SIGCOMM Computer Communication Review* 22.5 (1992), pp. 29–38. DOI: 10.1145/141809.141812. *(One citation on page 87.)*

[TV09] Stephen R. Tate and Roopa Vishwanathan. “Improving Cut-and-Choose in Verifiable Encryption and Fair Exchange Protocols Using Trusted Computing Technology”. In: *Data and Applications Security XXIII*. Springer. 2009, pp. 252–267. DOI: 10.1007/978-3-642-03007-9_17. *(One citation on page 183.)*

[TV13] Cristian Thiago Moecke and Melanie Volkamer. “Usable Secure Email Communications - Criteria and Evaluation of Existing Approaches”. In: *Information Management & Computer Security* 21.1 (2013), pp. 41–52. DOI: 10.1108/09685221311314419. *(One citation on page 13.)*

[TZ05] Amandeep Thukral and Xukai Zou. “Secure Group Instant Messaging Using Cryptographic Primitives”. In: *Networking and Mobile Computing*. Springer, 2005, pp. 1002–1011. DOI: 10.1007/11534310_105. *(One citation on page 34.)*

[UA13] David C. Uthus and David W. Aha. “The Ubuntu Chat Corpus for Multiparticipant Chat Analysis”. In: *AAAI Spring Symposium Series*. AAAI. 2013, pp. 99–102. URL: https://www.aaai.org/ocs/index.php/SSS/SSS13/paper/view/5706 (visited on 2021-04-24). *(One citation on page 456.)*

[UDB+15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. “SoK: Secure Messaging”. In: *Symposium on Security and Privacy (S&P)*. IEEE. 2015, pp. 232–249. DOI: 10.1109/SP.2015.22. *(Seven citations on pages iv, 1, 11, 26, 32, 37, and 67.)*

[UG15] Nik Unger and Ian Goldberg. “Deniable Key Exchanges for Secure Messaging”. In: *Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 1211–1223. DOI: 10.1145/2810103.2813616. *(Twenty-Seven citations on pages iv, 11, 23, 73, 80, 89, 96, 106, 107, 109, 113, 114, 118–120, 138, and 140.)*

[UG18] Nik Unger and Ian Goldberg. “Improved Strongly Deniable Authenticated Key Exchanges for Secure Messaging ”. In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2018.1 (2018), pp. 21–66. DOI: 10.1515/popets-2018-0003. *(Three citations on pages iv, 23, and 74.)*

[Ung15] Nik Unger. “Deniable Key Exchanges for Secure Messaging”. Master’s thesis. University of Waterloo, 2015. URL: https://hdl.handle.net/10012/9406 (visited on 2021-04-24). *(Three citations on pages iv, 11, and 138.)*

[Unr15] Dominique Unruh. “Non-Interactive Zero-Knowledge Proofs in the Quantum Random Oracle Model”. In: *Advances in Cryptology—EUROCRYPT*. Springer. 2015, pp. 755–784. DOI: 10.1007/978-3-662-46803-6_2. *(Two citations on pages 196 and 287.)*

[Van13a] Gilles Van Assche. *eXtended Keccak Code Package*. 2013. URL: https://github.com/XKCP/XKCP (visited on 2021-04-24). *(One citation on page 105.)*

[Van13b] Matthew D. Van Gundy. “Improved Deniable Signature Key Exchange for mpOTR”. 2013. URL: https://matt.singlethink.net/projects/mpotr/improved-dske.pdf (visited on 2021-04-24). *(One citation on page 35.)*

[VAS+17] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. “Measuring small subgroup attacks against Diffie-Hellman”. In: *Network and Distributed System Security Symposium (NDSS)*. Internet Society. 2017. DOI: 10.14722/ndss.2017.23171. *(One citation on page 205.)*

[VC12] Matthew D. Van Gundy and Hao Chen. “OldBlue: Causal Broadcast In A Mutually Suspicious Environment (Working Draft)”. 2012. URL: https://matt.singlethink.net/projects/mpotr/oldblue-draft.pdf (visited on 2021-04-24). *(One citation on page 34.)*

[VFO+18] Elham Vaziripour, Reza Farahbakhsh, Mark O'Neill, Justin Wu, Kent Seamons, and Daniel Zappala. "A Survey of the Privacy Preferences and Practices of Iranian Users of Telegram". In: *Workshop on Usable Security (USEC)*. 2018. DOI: 10.14722/usec.2018.23033. *(One citation on page 15.)*

[VGT+21] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, Isis Lovecruft, and Mike Hamburg. *The ristretto255 and decaf448 Groups*. Internet Draft. IETF, 2021. URL: https://tools.ietf.org/html/draft-irtf-cfrg-ristretto255-decaf448 (visited on 2021-04-24). *(One citation on page 370.)*

[VWO+18] Elham Vaziripour, Justin Wu, Mark O'Neill, Daniel Metro, Josh Cockrell, Timothy Moffett, Jordan Whitehead, Nick Bonner, Kent Seamons, and Daniel Zappala. "Action Needed! Helping Users Find and Complete the Authentication Ceremony in Signal". In: *Symposium on Usable Privacy and Security (SOUPS)*. USENIX. 2018, pp. 47–62. *(One citation on page 17.)*

[Wal08] Shabsi Walfish. "Enhanced Security Models for Network Protocols". PhD thesis. New York University, 2008. *(Seven citations on pages 23, 73, 116, and 120.)*

[Wei19] Matthew Weidner. "Group Messaging for Secure Asynchronous Collaboration". Master's thesis. University of Cambridge, 2019. *(Three citations on pages 27 and 67.)*

[WGH+19] Justin Wu, Cyrus Gattrell, Devon Howard, Jake Tyler, Elham Vaziripour, Kent Seamons, and Daniel Zappala. ""Something isn't secure, but I'm not sure how that translates into a problem": Promoting autonomy by designing for understanding in Signal". In: *Symposium on Usable Privacy and Security (SOUPS)*. USENIX. 2019. *(One citation on page 17.)*

[WGL00] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. "Secure Group Communications Using Key Graphs". In: *IEEE/ACM Transactions on Networking* 8.1 (2000), pp. 16–30. DOI: 10.1109/90.836475. *(Four citations on pages 49, 60, 61, and 292.)*

[WHA99] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. *Key Management for Multicast: Issues and Architectures*. RFC 2627. IETF, 1999, pp. 1–23. URL: https://tools.ietf.org/html/rfc2627 (visited on 2021-04-24). *(One citation on page 49.)*

[Wie03] Michael J. Wiener. *Safe Prime Generation with a Combined Sieve*. Tech. rep. 2003/186. Cryptology ePrint Archive, 2003. URL: https://eprint.iacr.org/2003/186 (visited on 2021-04-24). *(One citation on page 215.)*

[WIRE15] WIRE. *Wire: Modern communication, full privacy*. 2015. URL: https://wire.com/ (visited on 2021-04-24). *(One citation on page 2.)*

[WLL13] Chang-Ji Wang, Wen-Long Lin, and Hai-Tao Lin. "Design of An Instant Messaging System Using Identity Based Cryptosystems". In: *International Conference on Emerging Intelligent Data and Web Technologies*. IEEE. 2013, pp. 277–281. DOI: 10.1109/EIDWT.2013.53. *(One citation on page 30.)*

[WMS+09] Qianhong Wu, Yi Mu, Willy Susilo, Bo Qin, and Josep Domingo-Ferrer. "Asymmetric Group Key Agreement". In: *Advances in Cryptology—EUROCRYPT*. Springer. 2009, pp. 153–170. DOI: 10.1007/978-3-642-01001-9_9. *(One citation on page 52.)*

[WMZW11] Shangping Wang, Rui Ma, Yaling Zhang, and Xiaofeng Wang. "Ring signature scheme based on multivariate public key cryptosystems". In: *Computers & Mathematics with Applications* 62.10 (2011), pp. 3973–3979. DOI: 10.1016/j.camwa.2011.09.052. *(One citation on page 89.)*

[WQZ+11] Qianhong Wu, Bo Qin, Lei Zhang, Josep Domingo-Ferrer, and Oriol Farras. "Bridging Broadcast Encryption and Group Key Agreement". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2011, pp. 143–160. DOI: 10.1007/978-3-642-25385-0_8. *(One citation on page 52.)*

[WRLP94] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. "The Cognitive Walkthrough Method: A Practitioner's Guide". In: *Usability inspection methods*. John Wiley & Sons, Inc., 1994, pp. 105–140. DOI: 10.5555/189200.189214. *(One citation on page 12.)*

[WT99] Alma Whitten and J. Doug Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". In: *Security Symposium*. USENIX, 1999. *(One citation on page 12.)*

[WWX14] Weiqiang Wen, Libin Wang, and Min Xie. *One-Round Deniable Key Exchange with Perfect Forward Security*. Tech. rep. 2014/904. Cryptology ePrint Archive, 2014. URL: https://eprint.iacr.org/2014/661. *(One citation on page 23.)*

[WZ18] Justin Wu and Daniel Zappala. "When is a Tree Really a Truck? Exploring Mental Models of Encryption". In: *Symposium on Usable Privacy and Security (SOUPS)*. USENIX. 2018, pp. 395–409. *(One citation on page 15.)*

[XQL13] Hu Xiong, Zhiguang Qin, and Fagen Li. "A Taxonomy of Ring Signature Schemes: Theory and Applications". In: *IETE Journal of Research* 59.4 (2013), pp. 376–382. DOI: 10.4103/03772063.2013.10876518. *(One citation on page 89.)*

[Yao76] Andrew Chi-Chih Yao. "On the Evaluation of Powers". In: *Journal on Computing* 5.1 (1976), pp. 100–103. DOI: 10.1137/0205008. *(Two citations on pages 234 and 236.)*

[YK07] Chung-Huang Yang and Tzong-Yih Kuo. "The Design and Implementation of a Secure Instant Messaging Key Exchange Protocol". 2007. URL: http://security.nknu.edu.tw/psnl/publications/2007Technology_SIMPP.pdf (visited on 2021-04-24). *(Two citations on pages 31 and 33.)*

[YKAL08] Chung-Huang Yang, Tzong-Yih Kuo, TaeNam Ahn, and Chia-Pei Lee. "Design and Implementation of a Secure Instant Messaging Service based on Elliptic-Curve Cryptography". In: *Journal of Computers* 18.4 (2008), pp. 31–38. *(One citation on page 31.)*

[YLL+17] Zheng Yang, Chao Liu, Wanping Liu, Daigu Zhang, and Song Luo. "A new strong security model for stateful authenticated group key exchange". In: *International Journal of Information Security* (2017), pp. 1–18. DOI: 10.1007/s10207-017-0373-1.
(One citation on page 51.)

[YLP11] Taek-Young Youn, Changhoon Lee, and Young-Ho Park. "An efficient non-interactive deniable authentication scheme based on trapdoor commitment schemes". In: *Computer Communications* 34.3 (2011), pp. 353–357. DOI: 0.1016/j.comcom.2010.03.028.
(One citation on page 23.)

[YO07] Kazuki Yoneyama and Kazuo Ohta. "Ring Signatures: Universally Composable Definitions and Constructions". In: *Information and Media Technologies* 2.4 (2007), pp. 1038–1051. DOI: 10.11185/imt.2.1038. *(One citation on page 92.)*

[YT10] Guomin Yang and Chik How Tan. "Dynamic Group Key Exchange Revisited". In: *International Conference on Cryptology and Network Security (CANS)*. Springer. 2010, pp. 261–277. DOI: 10.1007/978-3-642-17619-7_19. *(Two citations on pages 48 and 305.)*

[YYK+16] Kazuki Yoneyama, Reo Yoshida, Yuto Kawahara, Tetsutaro Kobayashi, Hitoshi Fuji, and Tomohide Yamamoto. "Multi-cast Key Distribution: Scalable, Dynamic and Provably Secure Construction". In: *International Conference on Provable Security*. Springer. 2016, pp. 207–226. DOI: 10.1007/978-3-319-47422-9_12. *(One citation on page 44.)*

[YZ13] Andrew Chi-Chih Yao and Yunlei Zhao. "OAKE: A New Family of Implicitly Authenticated Diffie-Hellman Protocols". In: *Conference on Computer and Communications Security (CCS)*. ACM. 2013, pp. 1113–1128. DOI: 10.1145/2508859.2516695. *(One citation on page 23.)*

[ZK02] Fangguo Zhang and Kwangjo Kim. "ID-Based Blind Signature and Ring Signature from Pairings". In: *Advances in Cryptology—ASIACRYPT*. Springer. 2002, pp. 533–547. DOI: 10.1007/3-540-36178-2_33. *(One citation on page 89.)*

[Zoom19] Zoom. *How Zoom's Unique Architecture Powers Your Video First UC Future*. 2019. URL: https://www.youtube.com/watch?v=5BMbsFqtD0A (visited on 2021-04-24).
(One citation on page 500.)

[ZWQD10] Lei Zhang, Qianhong Wu, Bo Qin, and Josep Domingo-Ferrer. "Identity-Based Authenticated Asymmetric Group Key Agreement Protocol". In: *International Computing and Combinatorics Conference*. Springer. 2010, pp. 510–519. DOI: 10.1007/978-3-642-14031-0_54. *(One citation on page 52.)*